How to explain Java 8 to Java 21 migration in an interview

In an interview, I would explain the migration in a structured and practical way, focusing on **what changed, why it was needed, and how I handled risks**.

I usually start by saying that the migration was done in **phases**, not as a single big jump. First, I upgraded the **JDK and build tools** like Maven or Gradle to support Java 21 and updated the compiler settings. I made sure all third-party libraries and frameworks were compatible with Java 21, especially Spring, Hibernate, and logging libraries, because outdated dependencies are the most common breaking point.

Next, I addressed **language and API changes**. Java 8 features like streams, lambdas, and Optional remained fully compatible, so there was no rewrite needed there. However, I replaced deprecated or removed APIs where required. For example, I reviewed usages of older reflection or internal JDK APIs and replaced them with supported alternatives. I also checked warnings introduced by newer compilers and fixed them proactively.

Then I focused on **performance and JVM changes**. Java 21 comes with improved garbage collectors and better memory handling. I validated GC behavior, tuned JVM options where needed, and removed old flags that are no longer supported. I also ran performance and load tests to ensure there were no regressions after the upgrade.

I also leveraged **newer Java features selectively**, not blindly. For example, where it made sense, I used features like improved switch expressions, records for simple DTOs, and better concurrency utilities, but only in new or refactored code to keep the migration safe and incremental.

Finally, I ensured **stability and backward compatibility** by running full regression testing, integration tests, and monitoring the application closely in lower environments before production rollout. I'd conclude by saying the migration improved **performance, security, long-term support, and maintainability**, while keeping business logic unchanged.

Java 11

Java 11 is a Long Term Support release and is widely used in enterprise systems. In interviews, I explain Java 11 as a release that focused on stability, performance, cleanup of old APIs, and a few very practical language improvements that reduce boilerplate code.

1. New String methods

Java 11 added several useful methods to the String class. isBlank checks whether a string is empty or contains only whitespace. strip removes leading and trailing whitespace correctly, including Unicode spaces, which is better than trim. lines converts a multiline string into a stream of lines. repeat allows repeating a string multiple times. These methods are commonly used in validation and formatting logic.
Example

```
String text = "  Hello  ";

text.isBlank();

text.strip();

text.lines().forEach(System.out::println);

text.repeat(2);
```

2. Standard HttpClient API

Java 11 introduced a built-in HttpClient API to make HTTP calls. Before this, developers relied on third-party libraries. The new API supports synchronous and asynchronous calls and HTTP/2, making it suitable for modern REST communication.

Example

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

    .uri(URI.create("https://example.com"))

    .build();

client.send(request, HttpResponse.BodyHandlers.ofString());
```

3. var in lambda parameters

Java 11 allows the use of var in lambda expressions. This improves readability and allows annotations on lambda parameters. It is useful when dealing with complex lambda logic.

Example

```
list.forEach((var item) -> System.out.println(item));
```

4. Removal of deprecated and internal APIs

Many old and unsafe APIs were removed or strongly discouraged in Java 11. This includes internal JDK APIs that should not be used in applications. This cleanup improves security and long-term maintainability and forces developers to use standard supported APIs.

5. ZGC and GC improvements

Java 11 introduced Z Garbage Collector as an experimental feature. ZGC focuses on very low pause times, even with large heaps. Java 11 also improved G1 GC, making it the default garbage collector with better performance.

Example

```
-XX:+UseZGC
```

6. Flight Recorder and Mission Control open sourced

Java Flight Recorder and Java Mission Control became open source and available in OpenJDK. These tools are used for monitoring, profiling, and diagnosing performance issues in production systems.

7. Single-file source-code execution

Java 11 allows running a single Java file without explicitly compiling it first. This is useful for quick testing and scripting.

Example

```
java HelloWorld.java
```

Java 17

Java 17 is a Long Term Support release and is considered a major modernization version of Java. In interviews, I explain Java 17 as a release that focused on **cleaner code, safer design, better pattern matching, and stronger platform security**, while keeping backward compatibility strong.

1. Records

Records are used to represent immutable data objects like DTOs. They automatically generate constructor, getters, equals, hashCode, and toString methods. This removes a lot of boilerplate code and makes classes easier to read and maintain. Records are best used when the class is only meant to carry data and not business logic.

Example

```
public record User(int id, String name) { }
```

2. Sealed classes and interfaces

Sealed classes allow developers to control which classes can extend or implement them. This improves design safety and makes the code more predictable. It is especially useful in domain modeling where only a fixed set of subclasses should exist.

Example

```
public sealed class Shape permits Circle, Rectangle { }

final class Circle extends Shape { }

final class Rectangle extends Shape { }
```

3. Pattern matching for instanceof

This feature simplifies type checking and casting. Earlier, we had to check the type and then cast it manually. Java 17 allows doing both in a single step, reducing boilerplate and making code cleaner.

Example

```
if (obj instanceof String s) {
   System.out.println(s.length());
}
```

4. Switch expressions (standardized)

Switch expressions allow switch to return a value and be used as an expression instead of just a statement. This makes conditional logic more concise and readable, especially when mapping values.

Example

```
String result = switch (status) {
   case 200 -> "OK";
   case 404 -> "NOT FOUND";
   default -> "ERROR";
};
```

5. Strong encapsulation of JDK internals

Java 17 strongly encapsulates internal JDK APIs. Code that depends on internal or unsupported APIs will fail fast. This improves security and forces applications to rely only on standard, supported APIs, which is very important for long-term maintenance.

6. New garbage collectors and performance improvements

Java 17 includes improvements to G1 GC and supports modern garbage collectors like ZGC and

Shenandoah. These collectors aim to reduce pause times and improve throughput, especially in large applications. Most applications see performance improvements without code changes.

7. Enhanced pseudo-random number generators
Java 17 introduces new interfaces and implementations for random number generation, allowing better control, performance, and reproducibility compared to the older Random class.
Example

```
RandomGenerator generator = RandomGenerator.getDefault();

int value = generator.nextInt();
```

8. Security enhancements
Java 17 removes weak cryptographic algorithms and strengthens TLS and security defaults. This makes applications more secure out of the box, which is important for enterprise and cloud environments.

Java 21

Java 21 is a Long Term Support release and is considered a major milestone, especially for **concurrency, performance, and modern language design**. In interviews, I explain Java 21 as a release that makes Java more scalable for cloud and microservices while keeping code simple and readable.

1. Virtual threads
Virtual threads are lightweight threads managed by the JVM instead of the operating system. Traditional threads are expensive and limited in number, but virtual threads allow creating millions of concurrent tasks without exhausting system resources. This is extremely useful for IO-heavy applications like REST APIs and microservices. The best part is that existing blocking code works without rewriting it in reactive style.
Example

```
Thread.startVirtualThread(() -> {
   System.out.println("Running in virtual thread");
});
```

2. Structured concurrency
Structured concurrency simplifies managing multiple concurrent tasks. Instead of manually creating and tracking threads, tasks are grouped into a single logical unit. If one task fails, others can be cancelled automatically. This makes concurrent code easier to read, debug, and maintain.
Example

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
   scope.fork(() -> task1());
   scope.fork(() -> task2());
   scope.join();
}
```

3. Pattern matching for switch
Java 21 makes pattern matching for switch fully stable. Switch can now work with types and patterns, not

just constants. This reduces long if-else chains and makes code safer and more expressive.
Example

```
switch (obj) {
   case String s -> System.out.println(s);
   case Integer i -> System.out.println(i);
   default -> System.out.println("Unknown");
}
```

4. Record patterns

Record patterns allow destructuring records directly while checking types. This makes code concise when working with immutable data objects. It works very well with pattern matching and improves readability.
Example

```
if (user instanceof User(int id, String name)) {
   System.out.println(name);
}
```

5. Sequenced collections

Java 21 introduces better support for ordered collections. Lists, sets, and maps that maintain order now provide methods like getFirst and getLast. This removes the need for manual index handling and improves code clarity.
Example

```
list.getFirst();
list.getLast();
```

6. Scoped values

Scoped values provide a safer and more controlled alternative to ThreadLocal. They allow passing contextual data across method calls without relying on thread-bound storage, which is especially important when using virtual threads.
Example

```
ScopedValue.runWhere(USER, currentUser, () -> process());
```

7. Garbage collector and performance improvements

Java 21 improves G1 and ZGC garbage collectors, reducing pause times and improving throughput. These improvements help applications scale better under high load without major tuning.

Java 8

1. What is Functional Interface?

A functional interface is an interface that has exactly one abstract method. This single abstract method represents the functional behaviour of the interface. Functional interfaces are the foundation of lambda expressions in Java. They can also contain default methods and static methods without breaking the rule of

one abstract method. The annotation FunctionalInterface is optional but recommended because it gives compile time error if more than one abstract method is added.

2. What are the functional interfaces that were already present before Java 8?
Runnable represents a task that runs in a separate thread. It has only one method run and does not return any value.
Callable is similar to Runnable but it returns a value and can throw checked exceptions. It is mainly used with ExecutorService.
Comparator is used to define custom sorting logic. It has one abstract method compare which compares two objects.
Comparable is used to define natural ordering of objects. It has one method compareTo which compares the current object with another object.

3. Can we extend one functional interface from another functional interface?
Yes one functional interface can extend another functional interface as long as the child interface does not introduce an additional abstract method. If it adds another abstract method then it will no longer be a functional interface. Default methods do not affect this rule.

4. What are all the functional interfaces introduced in Java 8?
Predicate

Predicate is used to test a condition and return true or false. It is mostly used in stream filtering.

Method name is test which takes one input and returns boolean.

Example using stream

```java
List<Integer> numbers = List.of(10, 15, 20, 25);

numbers.stream()
    .filter(n -> n % 2 == 0)
    .forEach(System.out::println);
```

Here filter internally calls the test method.

Function

Function is used to transform one value into another value. It is commonly used in map operations.

Method name is apply which takes one input and returns one output.

Example using stream

```java
List<Integer> numbers = List.of(2, 3, 4);

numbers.stream()
      .map(n -> n * n)
      .forEach(System.out::println);
```

Here map internally calls the apply method.

Consumer

Consumer is used to perform an action on the given input. It does not return anything.

Method name is accept which takes one input and returns nothing.

Example using stream

```java
List<String> names = List.of("A", "B", "C");

names.stream()
    .forEach(name -> System.out.println(name));
```

Here forEach internally calls the accept method.

Supplier

Supplier is used to provide or generate a value. It does not take any input.

Method name is get which returns a value.

Example

```java
Supplier<String> supplier = () -> "Java";
System.out.println(supplier.get());
```

UnaryOperator

UnaryOperator is a special type of Function where input and output types are same.

Method name is apply inherited from Function.

Example using stream

```java
List<Integer> numbers = List.of(1, 2, 3);

numbers.stream()
    .map(n -> n + 1)
    .forEach(System.out::println);
```

BinaryOperator

BinaryOperator is a special type of BiFunction where both inputs and output are of same type.

Method name is apply inherited from BiFunction.

Example using stream

```java
List<Integer> numbers = List.of(1, 2, 3, 4);

int sum = numbers.stream()
            .reduce(0, (a, b) -> a + b);

System.out.println(sum);
```

BiPredicate

BiPredicate takes two inputs and returns true or false.

Method name is test which takes two inputs and returns boolean.

Example

```java
BiPredicate<Integer, Integer> bp = (a, b) -> a + b > 10;

System.out.println(bp.test(5, 7));
```

BiFunction

BiFunction takes two inputs and returns one output.

Method name is apply which takes two inputs and returns one output.

Example

```java
BiFunction<String, String, String> bf =
        (name, city) -> name + " from " + city;

System.out.println(bf.apply("Ram", "Delhi"));
```

BiConsumer

BiConsumer takes two inputs and performs an action without returning anything.

Method name is accept which takes two inputs and returns nothing.

Example using stream

```java
Map<Integer, String> map = Map.of(1, "A", 2, "B");

map.forEach((key, value) ->
        System.out.println(key + " " + value));
```

5. What is lambda?

Lambda is a short form of writing an implementation of a functional interface. Instead of creating a class or anonymous inner class you directly write the logic. Lambda expression represents the implementation of the single abstract method of a functional interface.

6. What are the advantages and disadvantages of using lambda expression?

Advantages are reduced boilerplate code better readability for small logic and support for functional programming style.

Disadvantages are harder debugging stack traces are less clear and complex lambda expressions can reduce readability.

7. What is stream?

tream is a feature introduced in Java 8 that allows us to process collections of data in a functional and declarative way. A stream represents a sequence of elements coming from a data source like a list, set, or array, and it supports operations such as filtering, mapping, sorting, and aggregation. Stream does not store data and does not modify the original collection. It only processes the data.

Streams work using lazy execution, which means operations are not executed until a terminal operation like forEach or collect is called. This helps improve performance because only required elements are processed. Streams can also be processed sequentially or in parallel. In real applications, streams are used to write clean, readable, and efficient code for data processing.

8. What is method reference?

Method reference is a shorter and cleaner way of writing a lambda expression when the lambda only calls an existing method. Instead of writing the full lambda syntax, we directly refer to the method using the class name or object name. This improves code readability and makes the intention very clear. Method reference works with functional interfaces because the referenced method must match the abstract method signature of the functional interface.

In simple terms, if a lambda expression only does one thing and that thing is calling another method, then method reference is a better choice.

There are four types of method references.

First is static method reference. It is used when the method being called is static. For example, when printing elements of a list, instead of writing a lambda, we can write list.forEach(System.out::println). Here println is a static method reference and it matches the Consumer interface.

Second is instance method reference of a particular object. It is used when the method belongs to a specific object. For example, if we have a Printer object and want to print each element, we can write names.forEach(printer::print). The print method belongs to the printer object.

Third is instance method reference of an arbitrary object of a class. It is used when the method belongs to any object of a particular class. A common example is names.stream().map(String::toUpperCase). Here toUpperCase is called on each String object in the stream.

Fourth is constructor reference. It is used to refer to a constructor. For example, names.stream().map(Employee::new).toList() creates Employee objects using the constructor.

9. Stream methods

filter is an intermediate operation used to select elements based on condition.
map is an intermediate operation used to transform elements.
flatMap is an intermediate operation used to flatten nested structures.
sorted is an intermediate operation used to sort elements.
forEach is a terminal operation used to iterate elements.
collect is a terminal operation used to convert stream into list set or map.
reduce is a terminal operation used to combine elements into a single result.

filter

filter is used to select elements based on a condition. It internally uses Predicate.

Example

```
List<Integer> numbers = List.of(10, 15, 20, 25);


numbers.stream()
    .filter(n -> n > 15)
    .forEach(System.out::println);
```

map

map is used to transform each element into another form. It internally uses Function.

Example

```
List<Integer> numbers = List.of(1, 2, 3);


numbers.stream()
    .map(n -> n * 2)
    .forEach(System.out::println);
```

flatMap

flatMap is used to flatten nested data structures into a single stream.

Example

```
List<List<Integer>> list = List.of(
    List.of(1, 2),
    List.of(3, 4)
);


list.stream()
```

```
    .flatMap(l -> l.stream())
    .forEach(System.out::println);
```

sorted

sorted is used to sort elements in natural or custom order.

Example

```
List<Integer> numbers = List.of(30, 10, 20);


numbers.stream()
       .sorted()
       .forEach(System.out::println);
```

forEach

forEach is a terminal operation used to perform an action on each element. It internally uses Consumer.

Example

```
List<String> names = List.of("A", "C", "B");


names.stream()
     .forEach(name -> System.out.println(name));
```

collect

collect is used to convert a stream into a collection or another data structure.

Example

```
List<Integer> numbers = List.of(1, 2, 3);


List<Integer> result = numbers.stream()
                              .map(n -> n * 2)
                              .collect(Collectors.toList());


System.out.println(result);
```

reduce

reduce is used to combine all elements into a single result. It internally uses BinaryOperator.

Example

```
List<Integer> numbers = List.of(1, 2, 3, 4);
```

```
int sum = numbers.stream()
        .reduce(0, (a, b) -> a + b);

System.out.println(sum);
```

findFirst

findFirst is used to get the first element from the stream. It returns Optional.

Example

```
List<Integer> numbers = List.of(5, 10, 15);

Optional<Integer> first = numbers.stream()
                .filter(n -> n > 6)
                .findFirst();

System.out.println(first.get());
```

anyMatch

anyMatch is used to check whether any element matches a condition. It returns boolean.

Example

```
List<Integer> numbers = List.of(3, 5, 8, 9);

boolean result = numbers.stream()
            .anyMatch(n -> n % 2 == 0);

System.out.println(result);
```

10. Map vs FlatMap

Map and flatMap are both used to transform data in Stream API, but they are used in different situations. Map is used when one input element produces exactly one output element. It applies a function to each element and returns a stream of transformed elements. For example, if we have a list of numbers and we want to square each number, we use map because each number becomes one squared value.

FlatMap is used when one input element produces multiple output elements. It first converts each element into a stream and then flattens all those streams into a single stream. FlatMap is commonly used when working with nested collections like a list of lists. For example, if we have a list of lists of integers and want a single list of all integers, flatMap is used.

11. Stream vs Parallel Stream

Stream processes elements sequentially using a single thread. Each element is handled one after another in a predictable order. It is simple to understand, easy to debug, and works well for small to medium data sets or when order of execution matters. Sequential streams are generally preferred in most business applications because they are stable and easier to control.

Parallel stream processes elements concurrently using multiple threads from the common ForkJoinPool. The data is split into multiple parts and processed in parallel on different CPU cores. This can improve performance for large data sets and CPU intensive operations. However, parallel streams introduce overhead, can cause thread contention, and may lead to incorrect results if the operations are not thread safe.

12. What is CompletableFuture?

CompletableFuture is a class used for asynchronous and non blocking programming in Java. It allows a task to run in the background without blocking the main thread and lets us define what should happen after the task completes. Unlike traditional Future, CompletableFuture supports chaining multiple actions, handling exceptions, and combining results of multiple asynchronous tasks.

13. CompletableFuture vs Future

Future represents the result of an asynchronous computation, but it has several limitations. When we call get on a Future, the calling thread is blocked until the result is available. Future also does not support chaining multiple tasks or proper exception handling, which makes it difficult to build complex asynchronous flows.

CompletableFuture overcomes these limitations. It is non blocking and allows us to attach callback methods that run automatically when the task completes. We can chain multiple asynchronous operations, combine results from different tasks, and handle exceptions in a clean way. In real applications, Future is suitable for simple async tasks, while CompletableFuture is preferred for building scalable and responsive systems.

14. How to decide thread pool size?

For CPU bound tasks thread pool size should be close to number of CPU cores because more threads will cause context switching.
For I O bound tasks thread pool size can be higher than number of cores because threads spend time waiting for I O operations.

15. Intermediate vs Terminal operations

In Stream API, operations are divided into intermediate and terminal operations based on how they behave during stream processing. Intermediate operations are operations that return another stream and do not produce a final result. Examples are filter, map, flatMap, and sorted. These operations are lazy, meaning they do not execute immediately when they are called. They only define the processing steps and execution starts only when a terminal operation is applied.

Terminal operations are operations that produce a result or a side effect and end the stream pipeline. Examples are forEach, collect, reduce, findFirst, and anyMatch. When a terminal operation is called, the stream starts processing all intermediate operations in sequence and produces the final result. After a terminal operation, the stream cannot be reused.

16. How does Stream API improve performance

Stream API improves performance mainly through lazy execution, pipelined processing, and optional parallelism. Lazy execution means that intermediate operations like filter and map are not executed immediately. They run only when a terminal operation is called, and only on the elements that are actually needed. This avoids unnecessary computations.

Streams also use pipelined processing, where each element flows through all intermediate operations before the next element is processed. This reduces the number of iterations over the data. Additionally, Stream API supports parallel streams, which can divide work across multiple CPU cores for CPU intensive tasks. Because of these reasons, Stream API often provides better performance along with cleaner and more readable code.

17. Difference between Optional and null and why Optional exists

Null represents the absence of a value, but it provides no information or safety. Using null often leads to NullPointerException at runtime, which is one of the most common bugs in Java applications. With null, it is not clear whether a value can be absent or not unless clearly documented.

Optional was introduced to solve this problem. Optional is a container object that may or may not contain a value. It forces the developer to explicitly handle the absence of a value using methods like isPresent, orElse, or ifPresent. This makes the code safer and more readable. Optional exists to reduce NullPointerException, clearly express that a value may be missing, and encourage better coding practices in modern Java applications.

18. How does CompletableFuture improve async programming

CompletableFuture improves asynchronous programming by removing the need to block threads while waiting for results and by providing a clean way to define what should happen after an async task completes. With older approaches like Future, the thread must call get and wait, which reduces performance and scalability. CompletableFuture allows tasks to run in the background and lets us attach callback methods that automatically execute when the result is ready.

It also supports chaining multiple asynchronous operations using methods like thenApply and thenCompose, combining results from multiple async tasks, and handling exceptions in a structured way. Because of this, CompletableFuture helps build non blocking, responsive, and scalable applications, which is especially important in modern web and microservices based systems.

Collection

1. What is List in Java?

List is an ordered collection that allows duplicate elements. Each element has an index, starting from zero. It is used when you want to store elements in the same order you insert them.

Common List Implementations

ArrayList
Stores elements in a dynamic array. Fast for reading (get) and slow for inserting in the middle. Good for most use cases.

LinkedList
Stores elements as nodes linked to each other. Fast for add and remove operations in the middle. Slower for random access.

Vector
Similar to ArrayList but synchronized. It is thread-safe but slower. Not used much today.

2. What is Set in Java?
Set is a collection that does not allow duplicate elements. It is used when you want only unique values.

Common Set Implementations

HashSet
Stores elements using hash values. It does not maintain order. Very fast for add, remove, and search.

LinkedHashSet
Same as HashSet but maintains insertion order. Useful when you need uniqueness + preserved order.

TreeSet
Stores elements in sorted order (natural or custom). Slower than HashSet because it uses a tree structure.

3. What is Map in Java?
Map stores data in key-value pairs. Keys must be unique, but values can be duplicate. It is used for fast lookups using keys.

Common Map Implementations

HashMap
Stores data using hashing. Does not maintain order. Very fast and most commonly used.

LinkedHashMap
Maintains insertion order. Useful for caching and predictable iteration.

TreeMap
Maintains keys in sorted order. Useful when you need sorted key-based operations.

Hashtable
Thread-safe version of HashMap but slower. Mostly replaced by ConcurrentHashMap.

ConcurrentHashMap
Thread-safe and fast. Used in multithreaded applications.

4. Quick Comparison Table

List → Ordered, allows duplicates
Set → Unordered or ordered depending on implementation, no duplicates
Map → Key-value store, keys unique, values can repeat

ArrayList → Fast read, slow insert in middle
LinkedList → Fast insert/remove, slow read
HashSet → Fast, no duplicates, no order
TreeSet → Sorted set

LinkedHashSet → fast, preserves insertion order, slightly more memory.
HashMap → Fast, no order
TreeMap → Sorted map
LinkedHashMap → Maintains insertion order

5. ArrayList vs LinkedList

ArrayList and LinkedList are both implementations of the List interface but they are designed for different use cases. ArrayList is backed by a dynamic array, so it provides fast random access using index, which makes get operations very efficient. However, inserting or deleting elements in the middle of an ArrayList is slow because elements need to be shifted.

LinkedList is backed by a doubly linked list, so insertion and deletion operations are faster, especially in the middle of the list, because only references are updated. However, accessing an element by index is slow because it requires traversal from the beginning or end. In real applications, ArrayList is preferred in most cases because read operations are more common, while LinkedList is useful when frequent insertions and deletions are required.

6. What will happen if we make an ArrayList collection as final in Java?
If an ArrayList is declared final, you cannot reassign the reference to a new list, but you can still modify the list itself by adding, removing, or updating elements.

7. HashSet vs TreeSet:

HashSet and TreeSet are both implementations of the Set interface, but they differ in how they store and retrieve elements. HashSet stores elements using a hash table, so it does not maintain any order and provides very fast operations for add, remove, and search with average constant time performance. It allows one null element and is preferred when ordering is not required and performance is important.

TreeSet stores elements in a sorted order using a balanced tree structure, specifically a red black tree. It maintains natural ordering or custom ordering using a comparator. Operations like add and search take logarithmic time, which is slower than HashSet. TreeSet does not allow null elements because comparison is required. In real applications, HashSet is used for fast lookup, while TreeSet is used when sorted data is needed.

8. HashMap vs TreeMap:

HashMap and TreeMap are both implementations of the Map interface, but they are used for different purposes. HashMap stores entries using a hash table and does not maintain any order of keys. It provides very fast performance for put and get operations with average constant time complexity. HashMap allows one null key and multiple null values and is commonly used when fast access is required and ordering is not important.

TreeMap stores entries in a sorted order based on the natural ordering of keys or a custom comparator. Internally it uses a red black tree, so operations like put, get, and remove take logarithmic time. TreeMap

does not allow null keys because it needs to compare keys. In real applications, HashMap is preferred for performance, while TreeMap is used when sorted keys or range based operations are required.

9. How to create custom ArrayList which doesn't allow duplicate?

```
public class CustomArrayList extends ArrayList {

    @Override
    public boolean add(Object o) {
        if(this.contains(o)){
            return true;
        }else{
            return super.add(o);
        }
    }
```

10. Why Set doesn't allow duplicates?

Set implementations in Java internally use a Map. For example, HashSet uses a HashMap internally, and every element you insert into the Set is stored as a **key** in the map, with a constant dummy value (usually Boolean.TRUE). Since a Map cannot have duplicate keys, the Set automatically prevents duplicate elements.

TreeSet is different: it uses a TreeMap internally, where elements become keys in a sorted tree structure.

11. Comparable vs Comparator:

Comparable
Comparable is used when a class defines its **own natural sorting order**. The class implements Comparable and overrides the compareTo() method. This means the sorting logic is inside the object itself. It is useful when you have a single default sorting rule, like sorting by id or name.

Comparator
Comparator is used when you want to define **multiple or custom sorting rules** outside the class. It is implemented separately and passed to sorting methods. This allows sorting the same object in different ways, like sorting employees by name, salary, or age.

12. Fail -fast and Fail-safe Iterator:

A fail fast iterator immediately throws ConcurrentModificationException if the collection is structurally modified while iterating, except through the iterator's own remove method. This behavior helps detect bugs early. Iterators of collections like ArrayList, HashMap, and HashSet are fail fast. For example, if one thread iterates over an ArrayList and another thread modifies it, the iterator will fail immediately.

Fail safe iterators do not throw ConcurrentModificationException. They work on a copy of the original collection, so modifications do not affect the iterator. Because of this, fail safe iterators may not reflect the latest changes but they provide safer iteration in concurrent environments. Iterators of collections like ConcurrentHashMap and CopyOnWriteArrayList are fail safe. In simple terms, fail fast detects problems quickly, while fail safe avoids exceptions by working on a snapshot.

13. Why do we need ConcurrentHashMap if Hashtable is already synchronized?

Hashtable is synchronized, but it uses full table locking. This means the entire Hashtable is locked for every read and write. Because of this, only one thread can access it at a time, which makes it very slow in multithreaded applications.

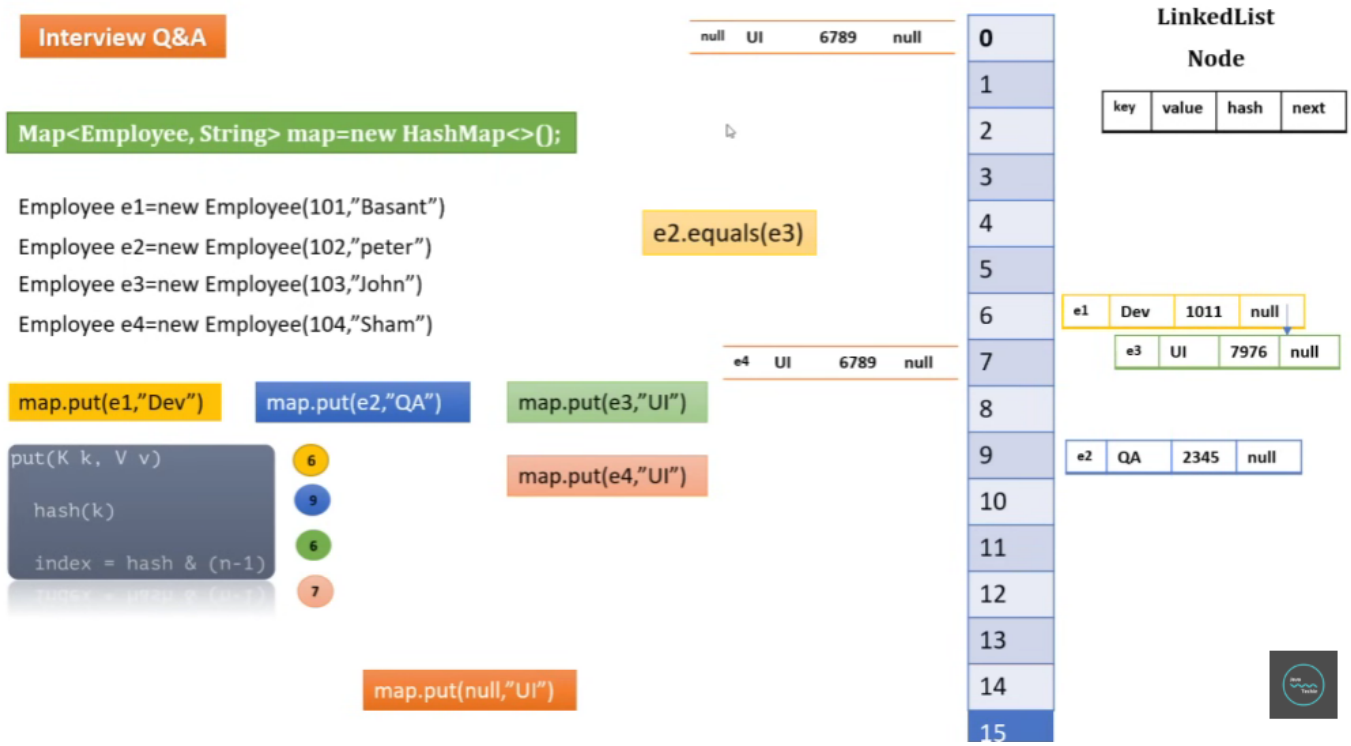
ConcurrentHashMap solves this by using partial locking (also called segment-level or bucket-level locking). Only small parts of the map are locked during updates, and reads happen almost without locking. This gives much higher performance in multi-threaded environments.

14. How do HashMap works internally?

Internally, HashMap uses an array of nodes, where each node contains a key, value, hash, and a reference to the next node (in case of collision). In Java 8 and above, these nodes can also form a **balanced tree** (Red-Black Tree) when collisions exceed a threshold.

When we insert a key-value pair:

- First, it calculates the hash code of the key.
- Then, it finds the index in the array using (n - 1) & hash.
- If the bucket is empty, it stores the node directly.
- If not, it checks for collision. If the key already exists (checked via equals()), the value is updated. If not, it adds the new node to the end of the list or tree."



15. If a key is null in HashMap then where that entry will store in map?

At the 0^{th} node.

16. How TreeMap Internally Works (Simple, Interview-Perfect Answer)

TreeMap stores key–value pairs in sorted order using a Red-Black Tree, which is a self-balancing binary search tree.

Every time you insert, search, or delete a key, TreeMap uses the tree structure to keep elements sorted and balanced.

Here is the step-by-step internal working:

- TreeMap stores keys in a Red-Black Tree
Each entry is stored as a Node containing:

- key
- value
- left child
- right child
- parent
- color (RED or BLACK)

Because it is sorted, your keys must be Comparable or you must provide a Comparator.

- Insertion uses Binary Search Tree logic
When you call:

treeMap.put(key, value);

TreeMap compares the key with the root:

- If new key < root → go left
- If new key > root → go right
- If key equals existing key → update value

Insertion always follows compareTo() or the provided Comparator.

17. How ConcurrentHashMap works internally?

- The map is divided into segments/buckets.
- When one thread updates bucket A, another thread can update bucket B without waiting.
- Read operations are non-blocking (no lock required in most cases).
- For updates, it uses CAS (Compare-And-Swap) + bucket-level locking.

****it doesn't allow duplicate keys or values because Null creates ambiguity in multi-threading ("key absent or value null?").**

18. How CopyOnWriteArrayList works internally?

CopyOnWriteArrayList is a thread-safe list that uses a *copy-on-write* strategy. It stores elements in a volatile array, and read operations never lock because the array is never modified in place. For every write—like add or remove—it takes a lock, creates a new copy of the entire array, applies the change, and then

replaces the old array reference. Its iterator is fail-safe because it works on a snapshot of the array. It gives very fast reads, but writes are expensive, so it's ideal for read-heavy, write-rare scenarios.

19 Can we use a custom class as a key in the HashMap and what is necessary for that?
Yes, a custom class can be used as a key in a HashMap. The class must override both the hashCode() and equals() methods to ensure the correct working of the hash-based data structure. Without these overrides, it may lead to incorrect behaviour, such as failing to retrieve the correct value.

20. Why is HashSet faster than ArrayList for search
HashSet is faster than ArrayList for search because of the data structure it uses internally. HashSet internally uses a HashMap to store elements. When we search for an element in a HashSet, the hashCode of the object is calculated and used to directly locate the bucket where the element should exist. Then equals is used only if needed to confirm the match. Because of this direct access, the average time complexity of search in HashSet is constant time.

ArrayList, on the other hand, stores elements in a sequential order. When searching in an ArrayList, it has to compare the target element with each element one by one using equals until it finds a match or reaches the end of the list. This results in linear time complexity and becomes slower as the list grows.

21. Collection type Hierarchy:

The Iterable interface is the root interface for all collection classes in Java. It is part of the java.lang package and provides the ability to traverse through a collection using an iterator. All the main interfaces of the collection framework (Collection, List, Set, Queue, etc.) extend the Iterable interface, which means that any class implementing these interfaces must provide an implementation of the iterator() method. This enables all collections to be **iterated** using a for-each loop or explicitly using an Iterator.

Multi-threading

1. How does multithreading improve performance and when does it fail?
Multithreading improves performance by allowing multiple tasks to run at the same time using different CPU cores. It is especially useful for IO bound tasks like database calls or network requests because while one thread is waiting, another thread can do useful work. However, multithreading fails when there are too many threads, which causes excessive context switching, shared resource contention, and synchronization overhead. In such cases, performance can actually become worse instead of better.

2. Difference between synchronized method and synchronized block
A synchronized method locks the entire object for the whole duration of the method execution. This means no other thread can access any synchronized method of that object at the same time. A synchronized block locks only a specific section of code or a specific object. Synchronized blocks are preferred because they reduce the scope of locking and improve performance by allowing more concurrency.

3. What is race condition? Real production example
A race condition occurs when multiple threads access and modify shared data at the same time, and the final result depends on the order of execution. A real production example is two threads updating the same bank account balance simultaneously. If both threads read the same balance and update it without synchronization, the final balance can be incorrect. Race conditions are prevented using synchronization, locks, or atomic variables.

4. What is volatile keyword and where is it used?

The volatile keyword ensures visibility of changes across threads. When a variable is declared volatile, any update made by one thread is immediately visible to other threads. It does not provide atomicity, so it cannot be used for compound operations like increment. Volatile is commonly used for flags or status variables, such as stopping a background thread safely.

5. Deadlock what is it and how do you prevent it?

Deadlock occurs when two or more threads are waiting for each other's locks indefinitely, so none of them can proceed. For example, Thread A holds Lock 1 and waits for Lock 2, while Thread B holds Lock 2 and waits for Lock 1. Deadlocks can be prevented by acquiring locks in a consistent order, avoiding nested locks, using timeouts, or using higher level concurrency utilities.

6. Difference between Runnable and Callable

Runnable represents a task that does not return any result and cannot throw checked exceptions. Callable represents a task that returns a result and can throw checked exceptions. Callable is used with ExecutorService and Future to get the result of asynchronous computation. In real applications, Callable is preferred when the task outcome is required.

7. Explain ExecutorService and thread pool internals

ExecutorService is a framework that manages a pool of reusable threads. Instead of creating a new thread for every task, tasks are submitted to the ExecutorService and executed by available threads from the pool. Internally, it uses a work queue to store tasks and worker threads to execute them. This improves performance, controls resource usage, and provides lifecycle management through shutdown methods.

8. How does Java handle thread safety in collections?

Java provides synchronized collections and concurrent collections. Synchronized collections lock the entire collection for each operation, which reduces performance. Concurrent collections like ConcurrentHashMap use fine grained locking or lock free techniques, allowing multiple threads to read and write concurrently with better scalability and performance.

9. How do you decide thread pool size for CPU-bound vs IO-bound tasks?

For CPU bound tasks, thread pool size should be close to the number of CPU cores because more threads cause unnecessary context switching. For IO bound tasks, thread pool size can be higher than the number of cores because threads spend time waiting for IO operations. The idea is to keep the CPU busy while other threads are waiting.

10. Explain wait, join, notify, and notifyAll

wait

wait is used when a thread wants to pause execution and release the lock until some condition is met. When a thread calls wait, it releases the monitor lock and goes into waiting state. It can only be called inside a synchronized block or method. A real example is a producer consumer scenario where the consumer waits if there is no data and resumes only when producer adds data and notifies it.

notify

notify is used to wake up one waiting thread. It does not release the lock immediately. The awakened thread will move from waiting state to runnable state only after the current thread releases the lock. notify is also called inside a synchronized block. It is generally used when only one waiting thread needs to continue execution.

notifyAll
notifyAll wakes up all waiting threads on the same object monitor. Like notify, it does not release the lock immediately. All awakened threads compete for the lock and one of them proceeds. notifyAll is safer than notify in most real applications because it avoids missed notifications and thread starvation, especially when multiple conditions are involved.

join
join is used when one thread wants to wait for another thread to complete execution. When a thread calls join on another thread, it enters waiting state until that thread finishes. join does not require synchronized block because it internally uses wait notify mechanism. A common example is the main thread waiting for worker threads to finish before proceeding.

11. What is ThreadLocal and where is it used?
ThreadLocal provides thread level storage, meaning each thread has its own independent copy of a variable. This avoids synchronization because data is not shared between threads. ThreadLocal is commonly used to store user context, transaction information, or request specific data in web applications and frameworks like Spring.

12. Difference between process and thread
A process is an independent program in execution, while a thread is a lightweight unit of execution inside a process. A process has its own separate memory space, resources, and address space, so processes are isolated from each other. Because of this isolation, communication between processes is slower and requires mechanisms like inter process communication.

A thread, on the other hand, runs inside a process and shares the same memory and resources with other threads of that process. This makes communication between threads faster and easier, but it also introduces concurrency issues like race conditions if not handled properly. Threads are cheaper to create and manage compared to processes, which is why multithreading is commonly used to improve performance.

13. What is context switching
Context switching is the process where the CPU stops executing one thread and starts executing another thread. When this happens, the CPU saves the current state of the running thread, such as its registers and program counter, and loads the state of the next thread. This allows multiple threads to share the CPU and gives the illusion of parallel execution, especially on single core systems.

14. What is daemon thread and how can we use it
A daemon thread is a background thread that runs to support user threads and performs auxiliary tasks. The JVM does not wait for daemon threads to finish execution. When all user threads complete, the JVM automatically terminates daemon threads. Because of this behavior, daemon threads are not suitable for important business logic.

Daemon threads are commonly used for background activities such as garbage collection, logging, monitoring, or cleanup tasks. For example, the garbage collector in Java runs as a daemon thread because it should stop when the application stops.

We can create a daemon thread by calling setDaemon true on a thread before calling start. Once a thread is started, its daemon status cannot be changed. In real applications, daemon threads are useful for background support work, but we must ensure that critical tasks are handled by non daemon user threads.

15. Difference between submit and execute methods
submit and execute are both used to submit tasks to an ExecutorService, but they differ in behavior and usage. execute is used to run a Runnable task and does not return any result. If an exception occurs during execution, it is thrown directly to the thread's uncaught exception handler, which can sometimes terminate the thread.

submit can be used with both Runnable and Callable and always returns a Future object. Using this Future, we can check task status, cancel the task, or get the result. Any exception thrown inside the task is captured and can be retrieved later through the Future. In real applications, submit is preferred when we need result tracking, exception handling, or task cancellation, while execute is suitable for fire and forget tasks.

SCENARIO-BASED:

1. A thread is stuck in waiting state. How do you identify and fix it?
If a thread is stuck in waiting state, the first thing I do is take a thread dump using tools like jstack or JVM monitoring tools. This shows which thread is waiting and on which lock or condition. Usually, the issue is a missing notify or notifyAll call, or improper use of wait inside synchronized blocks. To fix it, I ensure wait notify are used correctly, conditions are rechecked in a loop, and locks are properly released so the waiting thread can resume.

2. Two threads update the same bank account balance. How do you prevent inconsistency?
This is a classic race condition. To prevent inconsistency, I ensure that only one thread can update the balance at a time. This can be done using synchronized blocks, ReentrantLock, or atomic variables if the logic is simple. In real applications, account updates are usually wrapped inside a transactional or locked section so read modify write happens atomically.

3. Application slows down due to shared resource contention. Your approach?
First, I identify which resource is causing contention using thread dumps or profiling tools. Then I reduce the lock scope by synchronizing only the critical section instead of whole methods. I may also use ReadWriteLock, concurrent collections, caching, or split the shared resource to reduce contention. The goal is to minimize blocking between threads.

4. How do you safely stop a long-running background thread?
Threads should not be stopped forcefully. The safe way is to use thread interruption. I set an interrupt flag using interrupt and inside the thread logic I regularly check Thread.interrupted and exit gracefully. This allows cleanup of resources and prevents inconsistent state.

5. How do you schedule a task every 5 seconds?
I use ScheduledExecutorService. It provides methods like scheduleAtFixedRate or scheduleWithFixedDelay. This approach is better than using Timer because it handles exceptions properly and supports thread pooling, which makes it suitable for production systems.

6. Two threads must execute in strict order A then B. How?
To ensure execution order, I use coordination mechanisms like CountDownLatch or wait notify. For

example, Thread B waits on a latch until Thread A finishes its work and counts down. This guarantees ordering without busy waiting and is safe and readable.

7. ExecutorService rejecting tasks. Possible reasons?
Tasks are rejected when the thread pool is exhausted. Common reasons are all threads are busy, the task queue is full, or the maximum pool size is reached. Another reason is submitting tasks after shutdown. The rejection policy then decides how tasks are handled. Fix involves tuning pool size or queue capacity.

8. Shared counter gives random values. What's happening?
This is due to a race condition. Increment operation is not atomic and multiple threads update the counter at the same time. To fix this, I use AtomicInteger or synchronize the increment operation so updates are thread safe.

9. Need to wait for multiple threads to finish. What will you use?
I use CountDownLatch or ExecutorService with awaitTermination. CountDownLatch allows the main thread to wait until all worker threads complete. This is clean and commonly used in concurrent workflows.

10. Thread blocked on IO needs cancellation. What will you do?
For IO blocking, interrupt alone may not work. I close the underlying IO resource like socket or stream, which causes the blocked thread to exit. If possible, I use interruptible IO like NIO channels that respond to thread interruption.

11. Works locally but fails in production. What do you check first?
I first check environment differences such as CPU cores, memory limits, thread pool sizes, and load. Then I look for race conditions or missing synchronization that appear only under high concurrency. Logs and thread dumps from production help identify the root cause.

12. CPU-heavy task. More threads or fewer threads and why?
For CPU heavy tasks, fewer threads are better. Ideally, thread count should be close to the number of CPU cores. More threads cause context switching overhead and reduce performance instead of improving it.

13. Frequent deadlocks. How do you detect and eliminate them?
Deadlocks are detected using thread dumps or JVM tools that show circular lock dependencies. To eliminate them, I enforce consistent lock ordering, reduce nested locking, or use tryLock with timeout so threads can recover instead of waiting forever.

14. ExecutorService not shutting down after shutdown. Fix?
Shutdown only stops accepting new tasks but allows running tasks to finish. If it does not stop, I call awaitTermination with a timeout and then shutdownNow to interrupt running tasks. I also check for blocked threads or infinite loops inside tasks.

15. Share data between threads without synchronized. Options?
I use volatile variables for visibility, atomic classes for atomic operations, concurrent collections like ConcurrentHashMap, or ThreadLocal when data should be isolated per thread. These options reduce locking and improve performance while maintaining thread safety.

16. What problem does the Executor framework solve compared to creating threads manually?
The Executor framework solves the problem of uncontrolled thread creation and poor resource management. When we create threads manually, each task creates a new thread, which is expensive and can exhaust CPU and memory. Executor framework separates task submission from task execution. We

submit tasks, and the framework manages thread creation, reuse, scheduling, and lifecycle. This results in better performance, scalability, and easier maintenance in production systems.

17. What is Executor, ExecutorService, and ScheduledExecutorService?
Executor is a simple interface that executes a task without managing thread lifecycle. ExecutorService extends Executor and adds features like task submission, Future support, and shutdown control. ScheduledExecutorService extends ExecutorService and is used to run tasks after a delay or periodically. In real applications, ExecutorService and ScheduledExecutorService are commonly used for async and scheduled tasks.

18. How does ExecutorService internally manage threads and tasks?
ExecutorService internally uses a thread pool, a task queue, and worker threads. When a task is submitted, it is placed into a queue. If a thread is available, it picks up the task and executes it. If not, the task waits in the queue. Threads are reused instead of being destroyed, which reduces overhead and improves performance.

19. Difference between execute and submit?
execute is used to run a Runnable task and does not return any result or Future. If an exception occurs, it is thrown directly. submit can be used with Runnable or Callable and returns a Future object. Using Future, we can check task status, get results, or handle exceptions. submit is preferred when task monitoring or result is required.

20. What is a thread pool and why is it better than creating new threads?
A thread pool is a group of reusable threads managed by ExecutorService. Instead of creating a new thread for every task, tasks are executed by existing threads. This reduces thread creation overhead, improves performance, controls resource usage, and prevents system overload. Thread pools are essential in high concurrency applications.

21. Difference between newFixedThreadPool and newCachedThreadPool?
newFixedThreadPool creates a fixed number of threads and uses a queue to store extra tasks. It provides predictable resource usage. newCachedThreadPool creates threads as needed and reuses idle threads, but it has no upper limit. Cached thread pool can lead to resource exhaustion if tasks increase rapidly, so fixed thread pool is safer for production.

22. What happens internally when a task is submitted to ExecutorService?
When a task is submitted, ExecutorService checks if an idle thread is available. If yes, the task is assigned to that thread. If not, the task is placed in the queue. If the queue is full and thread limit allows, a new thread is created. Otherwise, the task is rejected based on the rejection policy. This controlled flow ensures stability.

23. How does Future work with ExecutorService?
Future represents the result of an asynchronous task. When a task is submitted using submit, a Future is returned. Using Future, we can check whether the task is completed, cancel it, or retrieve the result using get. This allows non blocking task submission and controlled result handling.

24. Difference between Runnable and Callable in Executor framework?
Runnable does not return a result and cannot throw checked exceptions. Callable returns a result and can throw checked exceptions. Callable is preferred when the task outcome is required or exception handling is important. Both are executed by ExecutorService.

25. How do you properly shut down an ExecutorService?

To properly shut down ExecutorService, first call shutdown to stop accepting new tasks and allow running tasks to finish. Then use awaitTermination to wait for completion. If tasks do not finish, call shutdownNow to interrupt running threads. This ensures graceful shutdown and avoids resource leaks.

26. Can you change thread pool size at runtime?

Yes, thread pool size can be changed at runtime, but only if we are using ThreadPoolExecutor directly. ExecutorService created using Executors utility methods does not expose methods to change the size. With ThreadPoolExecutor, we can change corePoolSize and maximumPoolSize using setter methods while the application is running. This is useful in production when load changes, for example increasing threads during peak traffic and reducing them later. However, changing pool size should be done carefully because increasing threads blindly can cause CPU contention and memory issues. In real systems, this is usually combined with monitoring and tuning rather than frequent dynamic changes.

27. Why should threads be reused instead of recreated?

Threads are expensive to create and destroy because they consume memory and involve interaction with the operating system. Creating a new thread for every task adds overhead and slows down the application under load. Reusing threads through a thread pool avoids this cost by keeping threads alive and assigning them new tasks when needed. This improves performance, reduces latency, and gives better control over resource usage. In production systems, thread reuse also prevents system overload and makes application behaviour more predictable under high concurrency.

Exception

1. Exception vs Error:

2. Throw vs Throws:

3. Checked vs Unchecked exception:

4. What is stacktrace?

Miscellaneous

1. I have the same method inside an abstract class and an interface. I implement both in a class. Which method gets invoked and why?

In this situation, the method from the abstract class is always invoked, not the interface method. This is because in Java, class methods have higher priority than interface default **methods** during method resolution.

When a class extends an abstract class and also implements an interface, Java follows a clear rule called class wins over interface. Even if the interface has a default method with the same signature, the JVM will always choose the method from the abstract class. The reason is that class inheritance existed before default methods were introduced in Java 8, so Java gives priority to class hierarchy to avoid ambiguity and to maintain backward compatibility.

For example, if an abstract class defines a concrete method and an interface defines a default method with the same name and parameters, and a class extends the abstract class and implements the interface, then calling that method on the object will execute the abstract class's method. The interface default method is

ignored unless the class explicitly overrides the method and chooses to call the interface version using InterfaceName.super.methodName, which is optional.

2. JVM, JRE, JDK:

The Java Virtual Machine (JVM): It is a virtual machine that allows Java applications to run on different platforms without modification. It takes the compiled Java bytecode and converts it into machine code that can be executed by the underlying hardware. The JVM handles memory management and execution of the program, offering platform independence by allowing the same Java program to run on Windows, macOS, or Linux without needing to be rewritten. The JVM is essential for running any Java program.

Java Runtime Environment (JRE): It is a software package that includes the JVM and the core libraries required to run Java applications. While the JVM handles the execution of the bytecode, the JRE provides the necessary libraries, files, and utilities to allow the application to run. The JRE is used by users who only need to run Java applications but are not involved in the development process.

Java Development Kit (JDK): It is a full development package that includes the JRE, the JVM, and tools required for developing Java applications. In addition to the JVM and libraries, the JDK provides a compiler (javac), a debugger, and other tools that help developers write, compile, and debug Java programs. The JDK is necessary for anyone who wants to develop Java software.

3. What is Enum and how we use it?

Enum is a type of class that allows developers to specify a set of predefined constant values. To create such a class we have to use the ***enum*** keyword. To iterate over all constants we can use the static ***values()*** method. Enums enable us to define members such as properties and methods like regular classes. Although it's a special type of class, we can't subclass it. An enum can, however, implement an interface. Another interesting advantage of *Enums* is that they are thread-safe and so they are popularly used as singletons.

```java
public enum Status {
    // Enum constants with multiple properties
    NEW(1, "Newly created"),
    IN_PROGRESS(2, "Currently in progress"),
    COMPLETED(3, "Task is completed"),
    FAILED(4, "Task failed");

    // Fields (properties) for each enum constant
    private final int code;
    private final String description;

    // Constructor to initialize the properties
    Status(int code, String description) {
        this.code = code;
        this.description = description;
    }

    // Getter method for 'code'
    public int getCode() {
        return code;
    }

    // Getter method for 'description'
    public String getDescription() {
        return description;
    }
}
```

4. Static binding vs Dynamic binding

Static binding and dynamic binding describe how Java decides which method to call. Static binding happens at compile time. The method call is resolved based on the reference type, not the actual object. Examples include method overloading, static methods, private methods, and final methods. Since the decision is made at compile time, static binding is faster and more predictable.

Dynamic binding happens at runtime. The method call is resolved based on the actual object type, not the reference type. This is the foundation of runtime polymorphism in Java. Method overriding uses dynamic binding. For example, if a parent class reference points to a child object, the child's overridden method is called at runtime.

5. Diamond problem

The diamond problem happens when a class inherits from two classes that have the same method, and the compiler does not know which method implementation to use. This creates ambiguity. Java avoids this problem by not allowing multiple inheritance with classes. A class in Java can extend only one class, so this ambiguity never occurs with classes.

However, the diamond problem can appear with interfaces in Java 8 and above because interfaces can have default methods. If a class implements two interfaces that have the same default method, the compiler forces the class to resolve the ambiguity by overriding the method. This is how Java handles the diamond problem safely.

For example, if two interfaces have the same default method, and a class implements both, the class must provide its own implementation. If one parent is a class and the other is an interface, the class method always wins. This rule is often explained in interviews as class over interface priority.

Example

```
interface A {
   default void show() {
      System.out.println("A");
   }
}


interface B {
   default void show() {
      System.out.println("B");
   }
}


class C implements A, B {
   @Override
   public void show() {
      A.super.show(); // or B.super.show()
   }
}
```

7. SOLID principles

SOLID is a set of five design principles that help us write clean, maintainable, and scalable object-oriented code. Interviewers usually expect not just definitions, but why they matter in real projects.

Single Responsibility Principle

This principle says a class should have only one reason to change, meaning it should do only one job. If a class handles multiple responsibilities, any change in one responsibility can break the others. In real projects, this makes code hard to maintain. For example, a UserService should not both save users and send emails. If email logic changes, user logic should not be affected.

Example

```
class UserService {
   void saveUser() { }
}
class EmailService {
```

```
    void sendEmail() { }
}
```

Open Closed Principle

This principle says a class should be open for extension but closed for modification. That means we should be able to add new behavior without changing existing code. This avoids breaking already tested code. In real systems, this is achieved using interfaces and polymorphism. For example, if we add a new payment method, we should not modify existing payment logic.

Example

```
interface Payment {
    void pay();
}
class CardPayment implements Payment {
    public void pay() { }
}
```

Liskov Substitution Principle

This principle says that a child class should be able to replace its parent class without breaking the application. In simple terms, wherever a parent object is expected, a child object should work correctly. Violating this causes unexpected behavior. For example, if a method expects a Bird, a Penguin should not break functionality like flying if flying is assumed.

Example

```
class Bird {
    void move() { }
}
class Sparrow extends Bird {
    void move() { }
}
```

Interface Segregation Principle

This principle says that clients should not be forced to depend on methods they do not use. Instead of one large interface, we should create smaller, specific interfaces. This makes implementations cleaner and avoids empty or unused methods. In real projects, this improves flexibility.

Example

```
interface Printer {
    void print();
}
interface Scanner {
    void scan();
```

```
}
```

Dependency Inversion Principle
This principle says high-level modules should not depend on low-level modules. Both should depend on abstractions. It also says abstractions should not depend on details. This is the foundation of Dependency Injection in Spring. For example, a service should depend on an interface, not a concrete implementation.
Example

```
interface MessageService {
   void send();
}
class Notification {
   private MessageService service;
   Notification(MessageService service) {
      this.service = service;
   }
}
```

8. JVM Memory Management
JVM memory management explains how Java allocates, uses, and frees memory while an application is running. The JVM divides memory into different areas so that object creation, method execution, and garbage collection can happen efficiently. Understanding this is important because many production issues like memory leaks, OutOfMemoryError, and performance problems are directly related to memory usage.

The main memory areas are Heap, Stack, Metaspace, Program Counter, and Native Method Stack. The Heap is used to store objects and class instances. It is shared across all threads. The heap is further divided into Young Generation and Old Generation. New objects are created in the Young Generation. If they survive multiple garbage collection cycles, they are moved to the Old Generation. Most garbage collection happens in the Young Generation because many objects are short-lived.

The Stack is used for method execution and stores local variables, method parameters, and call information. Each thread has its own stack, so stack memory is not shared. Because of this, stack operations are fast and thread-safe. StackOverflowError happens when too many nested method calls occur.

Metaspace stores class metadata such as class structure, methods, and constant pool. It replaces PermGen from Java 8 onwards and grows dynamically based on available system memory. If too many classes are loaded and memory is exhausted, Metaspace OutOfMemoryError can occur.

Garbage Collection is the automatic process of freeing memory by removing objects that are no longer reachable. JVM uses different garbage collectors like Serial, Parallel, G1, and ZGC depending on the Java version and configuration. The goal of GC is to balance low pause time and high throughput.

In real projects, good memory management means avoiding unnecessary object creation, closing resources properly, and monitoring heap usage. Tools like JVisualVM and GC logs are used to analyze memory behavior. The key interview line is JVM manages memory automatically, but developers must write memory-efficient code to avoid runtime issues.

9. Common causes of OutOfMemoryError

OutOfMemoryError occurs when the JVM cannot allocate more memory for an object even though garbage collection has already run. This usually indicates a memory leak, excessive memory usage, or incorrect JVM tuning. Understanding the cause is very important because OutOfMemoryError is a production-critical issue.

One common cause is memory leaks, where objects are no longer needed but still referenced. For example, storing objects in static collections like static HashMap and never removing them prevents garbage collection. Over time, heap memory fills up and causes OutOfMemoryError.

Another frequent cause is loading too much data into memory at once. For example, reading a huge database table into a List instead of using pagination or streaming. This quickly exhausts heap memory, especially in applications handling large datasets.

Improper JVM heap size configuration is also a major reason. If Xmx is set too low for the application workload, the JVM runs out of heap even though the code may be correct. This is common when applications move from local to production environments.

Metaspace OutOfMemoryError happens when too many classes are loaded and not unloaded. This is often caused by dynamic class loading, excessive use of proxies, or classloader leaks in application servers.

Thread-related memory issues can also cause OutOfMemoryError. Each thread consumes stack memory, so creating too many threads can exhaust native memory, leading to errors like unable to create new native thread.

10. Difference between == and .equals()

The difference between == and .equals() is a very common interview question and tests understanding of object comparison in Java. The == operator is used to compare references, meaning it checks whether two variables are pointing to the same memory location. It does not compare the actual content of the objects. So if two references point to the same object in memory, == returns true, otherwise false.

The .equals() method is used to compare the actual content or logical equality of objects. By default, the equals method from the Object class behaves the same as ==, meaning it compares references. However, many classes like String, Integer, and custom entity classes override equals to compare values instead of memory addresses.

For example, if we create two String objects with the same value using new keyword, == will return false because they are different objects in memory, but .equals() will return true because the content is the same. In real projects, we almost always use .equals() when comparing objects like IDs, names, or business data. The key interview point is that == compares memory reference, and .equals() compares content, provided it is overridden properly.

11. Explain the use of the Finalize() method of the garbage collector.

The Finalize() method is called by the garbage collector before collecting any object that is eligible for the garbage collector. The Finalize() method is used to get the last chance to object to cleaning and free remaining resource.

12. What is a memory leak, and how does it affect garbage collection?

A **memory leak** occurs when a program allocates memory for objects but fails to release it back to the system when those objects are no longer needed. This can lead to increased memory consumption over time, as unused objects remain in memory, preventing the garbage collector from reclaiming that space.

13. Serialization and Deserialization

Serialization is the process of converting a Java object into a byte stream so that it can be stored in a file, sent over a network, or cached. Deserialization is the reverse process, where the byte stream is converted back into a Java object. In real applications, this is commonly used for sending objects between systems, saving object state, or enabling distributed communication. Java provides built-in support for this using the Serializable interface.

To make an object serializable, the class must implement Serializable. This is a marker interface, meaning it has no methods, but it tells the JVM that the object can be converted into a byte stream. During serialization, Java writes the object's state, not its behavior. Static variables are not serialized because they belong to the class, not the object, and transient variables are skipped intentionally, usually for security or performance reasons.

Example

```
import java.io.Serializable;


class User implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private transient String password;


   User(String name, String password) {
      this.name = name;
      this.password = password;
   }
}
```

Serialization is done using ObjectOutputStream, and deserialization is done using ObjectInputStream. In practice, this might happen when saving data to disk or sending objects between microservices using messaging.

Example

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("user.ser"));
out.writeObject(user);
out.close();


ObjectInputStream in = new ObjectInputStream(new FileInputStream("user.ser"));
```

```
User u = (User) in.readObject();

in.close();
```

14. Ambiguity problem

The ambiguity problem in Java occurs when the compiler cannot decide which method or implementation to choose because multiple options match equally well. In simple terms, Java gets confused about which method to call, and as a result, compilation fails. This usually happens in scenarios like method overloading, multiple inheritance through interfaces, or default methods.

A very common example is method overloading ambiguity. If two overloaded methods have similar parameter types and we pass null or a value that matches both, the compiler cannot decide which method is the best match. In such cases, Java throws a compile-time error instead of guessing, because guessing could lead to unpredictable behavior.

Example

```
void test(String s) { }

void test(Integer i) { }


// test(null); // compile-time error due to ambiguity
```

Another important and frequently asked case is ambiguity with default methods in interfaces. If a class implements two interfaces that both have the same default method, Java does not know which default implementation to use. To resolve this ambiguity, Java forces the implementing class to override the method explicitly. This is how Java safely avoids ambiguity at runtime.

Example

```
interface A {
   default void show() { }
}


interface B {
   default void show() { }
}


class C implements A, B {
   public void show() {
      A.super.show();
   }
}
```

15. When a class has both void f(String s) and void f(Object s), and you call the method like f("kj"), the f(String s) method will be invoked.

In Java, method overloading resolution happens at compile time. The compiler always tries to find the most specific method that matches the argument type. A String literal like "kj" is of type String, not Object. Since String is a subclass of Object, both methods are technically valid, but f(String s) is more specific than f(Object s).

Because of this, the compiler chooses f(String s) without any ambiguity. This follows Java's rule that more specific parameter types are preferred over more general ones during method overloading.

Example

```
class Test {
   void f(String s) {
      System.out.println("String version");
   }

   void f(Object o) {
      System.out.println("Object version");
   }

   public static void main(String[] args) {
      Test t = new Test();
      t.f("kj");
   }
}
```

Output

String version

16. There is a method void f(String s){ sout("Hello"), if i call like f(null) what will happen?

When there is only one method void f(String s) and you call it as f(null), the method will be invoked successfully, and "Hello" will be printed. There will be no compile-time error and no runtime exception, as long as you do not use s inside the method.

In Java, null is a valid value for any reference type, including String. Since there is only one matching method and its parameter type is String, the compiler has no ambiguity and directly binds the call to f(String s). Java does not throw any error just because a parameter is null.

The method body executes normally. An exception occurs only if we try to use the variable s, such as calling s.length() or s.toUpperCase(). In that case, a NullPointerException will be thrown at runtime.

Example

```
void f(String s) {
   System.out.println("Hello");
}

f(null); // prints Hello
```

If we change the method to use s, then the behavior changes.

Example

```
void f(String s) {
   System.out.println(s.length());
}

f(null); // NullPointerException at runtime
```

17. Reactive programming vs Non-reactive programming in Java

In Java, the main difference between reactive and non-reactive programming is how threads and I O operations are handled.

Non-reactive programming follows a blocking and synchronous model. When a request comes in, one thread is assigned to it. If that thread makes a database call, REST call, or file read, it waits until the operation finishes. During this waiting time, the thread is blocked and cannot do anything else. This model is very easy to understand and debug, and it works well for applications with low or moderate traffic. Traditional Java approaches like Spring MVC, Servlets, JDBC, and RestTemplate are non-reactive. The main problem appears under high load, where many blocked threads can exhaust the thread pool and slow down the system.

Reactive programming in Java follows a non-blocking and asynchronous model. Instead of waiting for I O operations to complete, the thread is released immediately and reused for other requests. When the result is ready, the framework notifies the application and continues processing. Reactive programming is built on event streams and asynchronous data flow. In Java, this is implemented using Reactive Streams, Project Reactor, and frameworks like Spring WebFlux. Instead of returning objects directly, methods return reactive types like Mono or Flux.

Example

```
// Non reactive
User user = userService.getUser(id);

// Reactive
Mono<User> userMono = userService.getUser(id);
```

A key concept in reactive programming is backpressure, which allows the consumer to control how much data it can handle. This prevents memory overload and makes systems more stable under heavy load. Non-reactive programming does not support backpressure.

In real projects, non-reactive Java is preferred when the application is simple, business logic is heavy, and traffic is predictable. Reactive Java is preferred for high-concurrency, I O-heavy systems like APIs, streaming platforms, and microservices handling thousands of concurrent requests.