1. What is the difference between Comparable and Comparator?

- **Comparable**: An interface used to define the natural ordering of objects. The class itself implements this interface and provides the compareTo() method. It allows only a single ordering for a class.

- **Comparator**: An interface used to define custom orderings. It is implemented by a separate class and provides the compare() method. Multiple different orderings can be created for a class using different comparator implementations.

Player.java

```java
public class Player {
    private int ranking;
    private String name;
    private int age;

    // constructor, getters, setters
}
```

PlayerSorter.java

```java
public static void main(String[] args) {
    List<Player> footballTeam = new ArrayList<>();
    Player player1 = new Player(59, "John", 20);
    Player player2 = new Player(67, "Roger", 22);
    Player player3 = new Player(45, "Steven", 24);
    footballTeam.add(player1);
    footballTeam.add(player2);
    footballTeam.add(player3);

    System.out.println("Before Sorting : " + footballTeam);
    Collections.sort(footballTeam);
    System.out.println("After Sorting : " + footballTeam);
}
```

This leads to compile time error.

Using comparable:

```java
public class Player implements Comparable<Player> {

    // same as before

    @Override
    public int compareTo(Player otherPlayer) {
        return Integer.compare(getRanking(), otherPlayer.getRanking());
    }

}
```

Using comparator:

```java
public class PlayerRankingComparator implements Comparator<Player> {

    @Override
    public int compare(Player firstPlayer, Player secondPlayer) {
        return Integer.compare(firstPlayer.getRanking(), secondPlayer.getRanking());
    }

}
```

```java
public class PlayerAgeComparator implements Comparator<Player> {

    @Override
    public int compare(Player firstPlayer, Player secondPlayer) {
        return Integer.compare(firstPlayer.getAge(), secondPlayer.getAge());
    }

}
```

```
PlayerRankingComparator playerComparator = new PlayerRankingComparator();
Collections.sort(footballTeam, playerComparator);
```

Now let's run our *PlayerRankingSorter to* see the result:

```
Before Sorting : [John, Roger, Steven]
After Sorting by ranking : [Steven, John, Roger]
```

If we want a different sorting order, we only need to change the *Comparator* we're using:

```
PlayerAgeComparator playerComparator = new PlayerAgeComparator();
Collections.sort(footballTeam, playerComparator);
```

Now when we run our *PlayerAgeSorter,* we can see a different sort order by *age:*

```
Before Sorting : [John, Roger, Steven]
After Sorting by age : [Roger, John, Steven]
```

2. What is the difference between List and Set?

- **List**: A collection that allows duplicates and maintains the insertion order. It is index-based and elements can be accessed via their index.
- **Set**: A collection that doesn't allow duplicates and doesn't maintain any specific order (although specific implementations like LinkedHashSet do). It is used when uniqueness is a priority.

3. 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 behavior, such as failing to retrieve the correct value.

4. What is the purpose of the equals() **method?**

The equals() method is used to compare two objects for equality. It checks whether two objects

are logically "equal", rather than checking for reference equality (==). The method should be overridden in a custom class to define meaningful equality based on object properties.

---

5. 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.

---

6. What is the difference between throw and throws?

- throw: Used to explicitly throw an exception in code.
- throws: Declares the exceptions that a method might throw. It is part of the method signature, indicating that the method could potentially throw certain checked exceptions.

---

7. What will happen in a method with return type Integer, with try-catch-finally blocks raising exceptions and having different return statements?
In this case, the finally block will execute after the try and catch blocks, and if the finally block contains a return statement, it will override any return value from the try or catch blocks. So, the method will return 3 from the finally block, regardless of what happens in try or catch.

---

8. Difference between Runnable and Callable interfaces?

Runnable is the core interface provided for representing multithreaded tasks, and Java 1.5 provided Callable as an improved version of Runnable.

Runnable interface	Callable interface
It is a part of *java.lang* package since Java 1.0	It is a part of the *java.util.concurrent* package since Java 1.5.
It cannot return the result of computation.	It can return the result of the parallel processing of a task.
It cannot throw a checked Exception.	It can throw a checked Exception.
In a runnable interface, one needs to override the run() method in Java.	In order to use Callable, you need to override the call()

9. What is Executor and ExecutorService?

Executor: Executor is an interface that provides a simple way to execute tasks asynchronously using the execute(Runnable command) method. It abstracts away the details of thread management.
ExecutorService: ExecutorService is a subinterface of Executor that adds more features, like submitting tasks with submit() methods that return Future objects, allowing you to retrieve results and manage task execution. It also provides methods for shutting down the service, like shutdown() and shutdownNow().

10. How do you test the API performance?
API performance can be tested using tools like JMeter, Postman, or Gatling. You can measure metrics such as response time, throughput, latency, and resource usage (CPU, memory) under various loads and stress conditions.

11. What is a Future object?
A Future represents the result of an asynchronous computation. It allows you to retrieve the result or check the completion status of the task at a later point. It provides methods like get(), isDone(), and cancel().

12. How does HashMap work internally?
i. Data Structure

> "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."

```
static class Node<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
}
```

ii. Storing a Value: put(K key, V value)

> "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."

iii. Retrieving a Value: get(Object key)

"The process is similar:

- Calculate the hash and find the index.

- Go to the bucket, and if multiple nodes exist, it uses equals() to find the right key."

iv. Collision Handling

"If two keys have the same index (due to same hash or hash collision), they are stored in a **LinkedList** (Java 7 and below) or **LinkedList that can be converted to a Tree** (Java 8 and above). When nodes in a bucket exceed 8 and capacity is above 64, the list is converted to a **Red-Black Tree** for faster lookup."

v. Resizing

"When the number of entries exceeds the threshold (capacity * load factor), the HashMap **resizes** (doubles in size) and **rehashes** the keys. This is an expensive operation."

---

13. How does Spring ensure atomicity in applications?

Spring ensures atomicity in applications through transaction management. Using the @Transactional annotation, Spring manages transactions automatically, ensuring that a series of operations either completes entirely or rolls back in case of failure.

---

14. If an application is failing due to a memory issue, how do you check for the issue in Java?

You can use tools like VisualVM, JConsole, or heap dumps to analyze memory usage. Checking for memory leaks, inefficient data structures, or over-retained objects by analyzing garbage collection logs or profiling memory consumption helps in identifying the issue.

---

15. Why is Stream lazy in nature?

Streams in Java are lazy because they don't process data immediately. Intermediate operations like filter(), map() only set up the pipeline, and the actual processing happens when a terminal operation like forEach(), collect() is called. This allows for performance optimizations, such as short-circuiting and minimizing computations.

16. What are the different intermediate and terminal methods for the Stream API?

- **Intermediate methods**: filter(), map(), sorted(), limit(), skip(), etc.
- **Terminal methods**: forEach(), collect(), reduce(), count(), findAny(), findFirst(), etc.

17. What are the design patterns used in Spring?

Some design patterns used in Spring include:

- **Singleton Pattern**: For creating a single instance of a bean.
- **Factory Pattern**: The BeanFactory and ApplicationContext are examples.
- **Proxy Pattern**: Used in Spring AOP.
- **Template Method Pattern**: Used in JdbcTemplate, RestTemplate.

18. How many design patterns do you know in Java?

Common design patterns in Java include:

- **Singleton**
- **Factory**
- **Abstract Factory**
- **Builder**
- **Prototype**
- **Adapter**
- **Decorator**
- **Observer**
- **Strategy**
- **Command**
- **Template Method**

19. What is the difference between @Component **and** @Service **annotation? Can we delete** @Component **in place of** @Service**?**

- @Component: A generic stereotype annotation for Spring-managed components.
- @Service: A specialization of @Component that indicates that the class performs some business logic.
- Yes, you can delete @Component in favor of @Service because @Service is a more specific version of @Component.

20. What is cyclic dependency in Spring?

A cyclic dependency occurs when two or more beans depend on each other, leading to a circular

reference that Spring cannot resolve during bean creation. This can lead to errors during application startup unless proxy-based dependency injection (like @Lazy) is used.

```java
@Component
public class A {
    @Autowired
    private B b;
}

@Component
public class B {
    @Autowired
    private A a;
}
```

```java
@Component
public class A {
    @Autowired
    @Lazy
    private B b;
}
```

20.a. Internal working of @Lazy:

Spring reads the @Lazy annotation on field B b.
It does not inject the real B into A immediately.
Instead, it injects a proxy object (like a shell) that acts like B but doesn't trigger its construction yet.

Spring creates A first (no problem, because it only injected a proxy for B).
Then it creates B and injects A into B as normal.

When any method of A is called Spring sees that the proxy for B is being used.

It now creates the real B bean (just-in-time).
Then it forwards the method call to the real object.

---

21. Why do we need to implement the Cloneable interface when the Object class has a clone() method?

The Object class has a clone() method, but if a class does not implement the Cloneable interface, calling clone() will throw a CloneNotSupportedException. The Cloneable interface is a marker interface that signals to the runtime that cloning is allowed for that class. Without it, the JVM assumes that the class should not be cloned, and it enforces this by throwing an exception.

---

22. Can I use the clone() **method without overriding it?**

Yes, you can use the clone() method without overriding it as long as your class implements the Cloneable interface. The default implementation in Object will perform a shallow copy. However, if your class has fields that refer to other objects, this shallow copy will only duplicate the references, not the actual objects. For a deep copy, you would need to override clone() and manually clone the referenced objects.

---

23. Which All kinds of Modifiers Are Available in Java and What Is Their Purpose?

Access Modifiers:

public: The member is accessible from any other class.

protected: The member is accessible within its own package and by subclasses.

default (no modifier): The member is accessible only within its own package.

private: The member is accessible only within its own class.

Non-Access Modifiers:

static: Static fields or methods are class members, whereas non-static ones are object members. Class members don't need an instance to be invoked. They are called with the class name instead of the object reference name**.**

final: we have the *final* keyword. We can use it with fields, methods, and classes. When final is used on a field, it means that the field reference cannot be changed. So it can't be reassigned to another object. When final is applied to a class or a method, it assures us that that class or method cannot be extended or overridden.

abstract: When classes are abstract, they can't be instantiated. Instead, they are meant to be subclassed. When methods are *abstract*, they are left without implementation and can be overridden in subclasses.

synchronized: We can use it with the instance as well as with static methods and code blocks. When we use this keyword, we make Java use a monitor lock to provide synchronization on a given code fragment.

volatile: We can only use it together with instance and static fields. It declares that the field value must be read from and written to the main memory – bypassing the CPU cache.

transient: Prevents serialization of a variable when an object is serialized.

native: Indicates that a method is implemented in native code using JNI (Java Native Interface).

strictfp: Used to restrict floating-point calculations to ensure portability across different platforms.

---

24. Static Class: A static class is a class that cannot be instantiated, or have a variable created using the new operator. Static classes can only contain static members, such as static methods and constructors. *The top level class cannot be static in java, to create a static class we must create a nested class and then make it static.*

---

25. Methods of the Object class:

i. clone()

- **Purpose**: Creates and returns a copy (clone) of the object.
- **Note**: To use this method, a class must implement the Cloneable interface, otherwise, a CloneNotSupportedException is thrown.

ii. equals(Object obj)

- **Purpose**: Compares the object to another object for equality.
- **Default behavior**: It checks whether the two references point to the same object in memory (using ==).

```java
@Override
public boolean equals(Object obj) {
    // Check if the object is being compared with itself
    if (this == obj) {
        return true;
    }

    // Check if the object is an instance of Person
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    // Typecast obj to Person to compare data members
    Person other = (Person) obj;

    // Compare name and age for equality
    return name.equals(other.name) && age == other.age;
}
```

iii. finalize()

- **Purpose**: Called by the garbage collector before an object is destroyed.
- **Note**: It's used to release resources but is rarely used in modern Java due to non-deterministic behavior of garbage collection.

iv. getClass()

- **Purpose**: Returns the Class object associated with the object at runtime. This method is used to get metadata about the class of an object.

v. hashCode()

- **Purpose**: Returns an integer hash code value for the object.
- **Note**: If equals() is overridden, hashCode() must be overridden as well to maintain the contract between these two methods.

```java
@Override
public int hashCode() {
    return Objects.hash(name, age);
}
```

vi. notify()

- **Purpose**: Wakes up a single thread waiting on the object's monitor (used in synchronization). This method is called on an object that has been locked using synchronized.

vii. notifyAll()

- **Purpose**: Wakes up all the threads waiting on the object's monitor, rather than a single thread.

viii. toString()

- **Purpose**: Returns a string representation of the object.
- **Default behavior**: Returns the class name followed by the "@" symbol and the object's hash code in hexadecimal.

ix. wait()

- **Purpose**: Causes the current thread to wait until another thread invokes notify() or notifyAll() on the same object. This method is used for thread synchronization.

x. wait(long timeout)

- **Purpose**: Causes the current thread to wait for a specific amount of time (in milliseconds) before being notified or interrupted.

xi. wait(long timeout, int nanos)

- **Purpose**: Similar to wait(long timeout), but allows you to specify a more precise time by adding nanoseconds in addition to milliseconds.

---

26. Sequence to run java program:

- **Write Code: You write the** .java **file containing your Java source code.**

  No components are involved yet.

- **Compile Code**: The **Java Compiler** (javac), part of the **JDK**, compiles the .java file into bytecode, which is saved in a .class file.

  JDK is required for this compilation process.

- **Run Program**: The **JVM**, part of the **JRE**, runs the .class file through the following steps:

  Class Loader: Loads the bytecode into memory.
  Bytecode Interpreter: Interprets the bytecode and converts it into platform-specific machine code one instruction at a time.
  JIT Compiler (optional): During execution, the **JIT** compiler may optimize performance by compiling frequently used bytecode into native machine code.
  If **AOT** (Optional) compilation is used, it would compile the bytecode into machine code before execution, reducing the need for JIT during runtime.

---

27. 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.

---

28. 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;
    }
}
```

29. What Is a NullPointerException?

The NullPointerException is probably the most common exception in the Java world. It's an unchecked exception and thus extends RuntimeException. We shouldn't try to handle it.

This exception is thrown when we try to access a variable or call a method of a null reference, like when:

- invoking a method of a null reference

- setting or getting a field of a null reference

- checking the length of a null array reference

- setting or getting an item of a null array reference

- throwing null

30. What Are Two Types of Casting in Java? Which Exception May Be Thrown While Casting? How Can We Avoid It?

Upcasting: Casting a subclass object to a superclass reference. This is implicit and safe.

Downcasting: Casting a superclass reference back to a subclass object. This requires explicit casting and can throw exceptions if done incorrectly.

Animal animal = new Dog(); // Upcasting

Dog dog = (Dog) animal;   // Downcasting

To avoid this exception, you can use the instanceof operator to check the actual type of the object before casting.

if (animal instanceof Dog) {

    Dog dog = (Dog) animal;

} else {

    System.out.println("animal is not a Dog");

}

---

31. Why Is String an Immutable Class?

In Java, String is immutable, meaning once a string object is created, its value cannot be changed. Here's why:

1. **Security**: Strings are used in sensitive places, like file paths and network connections. Immutability ensures that once a string is created, it cannot be altered, preventing security risks like someone changing a database URL mid-execution.

2. **Thread Safety**: Since strings cannot be modified, they are inherently thread-safe. Multiple threads can use the same string object without needing synchronization, making programs more efficient in multithreaded environments.

3. **Memory Optimization (String Pool)**: Java maintains a **String pool** where identical string objects are reused to save memory. If strings were mutable, changing one string would affect all references pointing to the same object, breaking this optimization.

4. **Consistent Hashcode**: Strings are often used as keys in hash-based collections like **HashMap**. Immutability ensures the string's hashcode doesn't change, avoiding issues with data retrieval if the string were modified after being used as a key.

In summary, immutability makes String secure, efficient, thread-safe, and predictable.

---

32. Static Binding vs Dynamic Binding:

Binding refers to the process of associating a method call or variable reference with the corresponding method implementation or memory location. There are two primary types of binding:

Static Binding (Early Binding)

1. Definition: Static binding refers to the compile-time resolution of method calls. The method to be executed is determined at compile time based on the method signature and the reference type.

2. How It Works: The compiler resolves method calls based on the reference type. This type of binding is used for method calls involving:

- **Static methods**
- **Private methods**
- **Final methods**
- **Methods in the same class**

3. Performance: Static binding is generally faster because the method calls are resolved at compile time and do not require runtime decision-making.

```
class Example {

  static void display() {

    System.out.println("Static method");

  }

}
```

Example.display(); // Static binding

Dynamic Binding (Late Binding)

1. Definition: Dynamic binding refers to the runtime resolution of method calls. The method to be executed is determined at runtime based on the actual object type (not the reference type).

2. How It Works: The JVM determines the method implementation to call at runtime using the actual object instance. This type of binding is used for:

- Overridden methods in subclasses
- Polymorphism

3. Performance: Dynamic binding can be slower than static binding because the JVM has to determine the actual method to call at runtime. However, it allows for flexibility and polymorphism in object-oriented programming.

```
class Parent {

  void display() {

    System.out.println("Parent method");

  }
```

```
}
class Child extends Parent {

    @Override

    void display() {

        System.out.println("Child method");

    }

}
```

Parent obj = new Child();

obj.display(); // Dynamic binding: "Child method" is called

---

33. JIT and AOT:

JIT: Just In Time Compiler: The Just-In-Time (JIT) compiler is a component of the runtime environment that improves the performance of Java™ applications by compiling bytecodes to native machine code at run time.

AOT: Ahead Of Time Compiler: It used to compile byte code to native code prior to the execution by the JVM.

Comparison:

Feature	JIT (Just-In-Time)	AOT (Ahead-Of-Time)
Timing of Compilation	At runtime	Before execution
Platform Independence	Yes, compiles at runtime	No, platform-specific binary
Startup Time	Slower, due to runtime compilation	Faster, already compiled
Optimization	Runtime optimizations based on usage patterns	Precompiled, no runtime optimizations
Performance	Can improve as the program runs	Fast startup, but lacks adaptive optimizations

Notes: Java programs can run without JIT or AOT using interpretation by JVM causes slower execution. And also JIT and AOT can be used together, especially in JVMs like GraalVM.

---

34. What Is a Classloader?

The classloader is one of the most important components in Java. It's a part of the JRE.

Simply put, the classloader is responsible for loading classes into the JVM. We can distinguish three types of classloaders:

Bootstrap classloader – it loads the core Java classes. They are located in the <JAVA_HOME>/jre/lib directory

Extension classloader – it loads classes located in <JAVA_HOME>/jre/lib/ext or in the path defined by the java.ext.dirs property

System classloader – it loads classes on the classpath of our application

A classloader loads classes "on demand". It means that classes are loaded after they are called by the program. What's more, a classloader can load a class with a given name only once. However, if the same class is loaded by two different class loaders, then those classes fail in an equality check.

---

35. Java 8 Important Features:

Java 8 introduced several significant features that improved the language's functionality and performance. Here are the key features along with their explanations:

1. Lambda Expressions

Lambda expressions are anonymous functions (methods without a name) that can be used to define the implementation of an interface's method in a concise way.

- Syntax: `(parameters) -> expression` or `(parameters) -> { statements }`

- Example:

```java
List<String> names = Arrays.asList("John", "Jane", "Doe");
names.forEach(n -> System.out.println(n));  // Using lambda expression
```

Use Case: Reduces boilerplate code, especially in situations requiring single-method interfaces like event handling or list processing.

2. Functional Interfaces

An interface with only one abstract method is called a functional interface. Java 8 introduced a `@FunctionalInterface` annotation to indicate such interfaces.

- Example:

```java
@FunctionalInterface
```

```java
interface MyFunctionalInterface {

  void myMethod();

}
```

Common Functional Interfaces: `Runnable`, `Callable`, `Comparator`, `Predicate`, `Function`.

3. Stream API

The Stream API provides a functional approach to process collections of data. It supports operations like filtering, mapping, and reduction.

- Example:

```java
List<String> names = Arrays.asList("John", "Jane", "Doe");

names.stream()

    .filter(name -> name.startsWith("J"))

    .forEach(System.out::println);  // Output: John, Jane
```

Use Case: Efficiently process large amounts of data using operations such as map, filter, and reduce, without modifying the original data source.

4. Default Methods in Interfaces

Java 8 allows interfaces to have methods with a default implementation using the `default` keyword. This allows the addition of new methods in interfaces without breaking the existing implementations.

- Example:

```java
interface Vehicle {

  default void start() {

    System.out.println("Vehicle is starting");

  }

}
```

```
class Car implements Vehicle { }
```


Use Case: It supports backward compatibility and the ability to evolve interfaces.

5. Optional Class

`Optional` is a container object which may or may not contain a non-null value. It helps to avoid `NullPointerException` by enforcing explicit checks for the presence of a value.

- Example:

```java
Optional<String> name = Optional.ofNullable(null);
System.out.println(name.orElse("Unknown"));  // Output: Unknown
```


Use Case: Makes it easier to handle cases where a value might be `null`, thus avoiding `NullPointerException`.

6. Method References

Method references allow you to refer to methods directly without executing them. They provide a more readable alternative to lambda expressions when referring to existing methods.

- Syntax: `ClassName::methodName`

- Example:

```java
names.forEach(System.out::println);  // Using method reference
```


Use Case: Simplifies lambda expressions and improves code readability.

7. Date and Time API (java.time)

Java 8 introduced a new date and time API under the `java.time` package, which is thread-safe and much more comprehensive than the old `java.util.Date` and `Calendar` classes.

- Example:

```java
LocalDate today = LocalDate.now();

LocalTime now = LocalTime.now();

LocalDateTime currentDateTime = LocalDateTime.now();
```

Use Case: Provides a modern, easy-to-use date and time management library.

8. Collectors

Collectors are used in conjunction with streams to accumulate the results of a stream into various collection types or apply summarization.

- Example:

```java
List<String> filteredNames = names.stream()
    .filter(name -> name.startsWith("J"))
    .collect(Collectors.toList());
```

Use Case: Allows easy conversion of stream results into collections, strings, or custom aggregation results.

9. Base64 Encoding and Decoding

Java 8 introduced a built-in API for encoding and decoding using Base64, under the `java.util.Base64` class.

- Example:

```java
String encoded = Base64.getEncoder().encodeToString("hello".getBytes());

byte[] decodedBytes = Base64.getDecoder().decode(encoded);
```

Use Case: Easily encode and decode data, commonly used for transmitting binary data in text format, such as in emails or web APIs.

These are the major features of Java 8 that brought more functional and streamlined ways of coding, along with better performance and safety mechanisms.

---

36. Describe the Collections Type Hierarchy. What Are the Main Interfaces, and What Are the Differences Between Them?

Iterable Interface:

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.

Collection Interfce:

The root interface of the collections hierarchy (except Map). It defines common methods like add(), remove(), and iterator().

- **Subinterfaces**: List, Set, Queue.
- **Methods**: add(), remove(), size(), clear(), iterator(), etc

List Interface:

Represents an ordered collection (also known as a sequence) where duplicates are allowed. Elements can be accessed by their index.

- **Classes**: ArrayList, LinkedList, Vector.
- **Key Methods**: get(int index), set(int index, E element), add(E element), remove(int index).

Set Interface:

epresents a collection that does not allow duplicate elements. It's an unordered collection, meaning there's no guarantee of element order.

- **Subinterfaces**: SortedSet (extends Set), NavigableSet.
- **Classes**: HashSet, LinkedHashSet, TreeSet.
- **Key Methods**: add(E element), remove(Object o), contains(Object o), size().

Queue Interface:

Represents a collection designed for holding elements prior to processing. Typically follows FIFO (First-In-First-Out) order.

- **Subinterfaces**: Deque (Double-ended Queue).
- **Classes**: LinkedList, PriorityQueue.
- **Key Methods**: offer(E e), poll(), peek().

Map Interface:

Represents a collection of key-value pairs. It doesn't extend the Collection interface and allows keys to be unique but values can be duplicated.

- **Subinterfaces**: SortedMap, NavigableMap.
- **Classes**: HashMap, LinkedHashMap, TreeMap, Hashtable.
- **Key Methods**: put(K key, V value), get(Object key), remove(Object key), keySet(), values().

---

37. Describe Various Implementations of the Map Interface and Their Use Case Differences

The `Map` interface in Java has several implementations, each suited for different use cases based on their underlying data structures and performance characteristics.

HashMap is one of the most commonly used implementations, offering constant-time complexity for basic operations like `get()` and `put()`, making it ideal for scenarios where quick access to data is essential, such as caching or lookup tables. However, it does not maintain any order of its entries. On the other hand, LinkedHashMap extends `HashMap` and preserves the insertion order of its entries, making it suitable for applications where the order of iteration matters, such as when maintaining the order of items in a UI component.

TreeMap, which implements the `NavigableMap` interface, sorts its keys according to their natural ordering or a specified comparator. This makes it useful for scenarios that require a sorted representation of keys, such as in applications that need to maintain a sorted list of items for quick access.

Hashtable is a synchronized implementation of the `Map` interface, providing thread safety at the cost of performance. While it is largely considered obsolete in favor of more modern alternatives like `ConcurrentHashMap`, it can still be used in multi-threaded environments where a simple synchronized map is needed.

ConcurrentHashMap offers high concurrency and is designed for use in multi-threaded applications, allowing multiple threads to read and write to the map without locking the entire structure, making it a great choice for high-performance applications where thread safety is critical. Each of these implementations serves distinct use cases, allowing developers to choose the most appropriate one based on the specific requirements of their application regarding performance, ordering, and concurrency.

---

38. ArrayList vs LinkedList:

ArrayList is backed by a dynamic array, which means it provides fast random access to elements using an index. This results in constant time complexity (O(1)) for retrieving elements. However, adding or removing elements, especially in the middle of the list, can be slower (O(n)) because it may require shifting elements to maintain the array's order. ArrayList is best suited for scenarios where frequent access and iteration over the list are required, and the size of the list is relatively stable or grows predictably.

In contrast, LinkedList is implemented as a doubly linked list, where each element (node) contains a reference to the next and previous nodes. This structure allows for constant time complexity (O(1)) for adding and removing elements at both ends (head and tail) and also when inserting or deleting in the middle of the list, as it only involves adjusting a few references. However, accessing an element by index takes linear time (O(n)) because it requires traversing the list from the beginning or end. LinkedList is advantageous in situations where there are many insertions and deletions, particularly in the middle of the list, and when memory overhead is not a significant concern since each element requires additional space for the node references.

---

39. HashSet vs TreeSet:

HashSet and TreeSet are both implementations of the Set interface in Java, but they differ significantly in their underlying data structures, performance characteristics, and usage scenarios.

HashSet is backed by a hash table, which means it provides constant-time performance (O(1)) for basic operations like add(), remove(), and contains(), assuming the hash function distributes elements evenly. However, HashSet does not maintain any order among its elements; the iteration order is unpredictable and can change over time as elements are added or removed. This makes HashSet ideal for scenarios where fast access and membership checks are important, and the order of elements is not a concern.

On the other hand, TreeSet is implemented using a red-black tree, a self-balancing binary search tree. This structure allows TreeSet to maintain elements in their natural ordering (or according to a specified comparator) and provides a logarithmic time complexity (O(log n)) for basic operations like add(), remove(), and contains(). The iteration order in a TreeSet is predictable, as it reflects the sorted order of the elements. Consequently, TreeSet is more suitable for applications where sorting is required, such as maintaining a sorted list of items or performing range queries.

---

40. What Is the Purpose of the Throw and Throws Keywords?

The *throws* keyword is used to specify that a method may raise an exception during its execution. It enforces explicit exception handling when calling a method

The *throw* keyword allows us to throw an exception object to interrupt the normal flow of the program. This is most commonly used when a program fails to satisfy a given condition

---

41. What Is the Difference Between a Checked and an Unchecked Exception?

Checked exceptions are exceptions that are checked at compile time. The Java compiler forces the programmer to either handle these exceptions using a try-catch block or declare them in the method signature using the throws keyword.

Examples: IOException, SQLException, FileNotFoundException.

Unchecked exceptions are exceptions that occur at runtime and are not checked by the compiler. The programmer is not required to explicitly handle them or declare them in the method signature.

Examples: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException

---

42. Exception vs Error:

Exception: Represents issues that occur during program execution, which can typically be handled and recovered from. Examples include IOException, NullPointerException, and SQLException. These are usually caused by external conditions or logical errors in the program.

Error: Represents serious problems that are generally unrecoverable and cannot be handled by the program. Examples include OutOfMemoryError, StackOverflowError, and VirtualMachineError. These usually indicate issues with the JVM or system resources.

---

43. Exception Chaining:

Occurs when an exception is thrown in response to another exception. This allows us to discover the complete history of our raised problem:

eg.

try {

   task.readConfigFile();

} catch (FileNotFoundException ex) {

   throw new TaskException("Could not perform task", ex);

}

---

44. What Is a Stacktrace and How Does It Relate to an Exception?

A stack trace is a report that shows the sequence of method calls in the program at a specific point in time, usually when an exception occurs. It provides details about the method call hierarchy, helping to identify where the exception was thrown and what led up to it.

When an exception occurs, the stack trace shows the exact path (stack of method calls) that the program took before the exception was thrown. This includes:

- The **exception type** (e.g., NullPointerException, IOException).
- The **message** associated with the exception (if any).
- The **class and method** where the exception occurred.
- The **line number** where the exception was thrown.
- A list of method calls leading to the exception, starting from the most recent (top of the stack) and going back in time.

---

45. Why Would You Want to Subclass an Exception?

If the exception type isn't represented by those that already exist in the Java platform or to create custom exceptions that are specific to your application or domain.

---

46. Can You Throw Any Exception Inside a Lambda Expression's Body

In Java, you can throw exceptions inside a lambda expression's body, but there are specific rules depending on whether the exception is a checked or unchecked exception.

Unchecked Exceptions (RuntimeException and its Subclasses):

You can throw **unchecked exceptions** (like NullPointerException, ArithmeticException, etc.) without any special handling because these exceptions are not checked at compile-time.

```java
Runnable r = () -> {
    throw new NullPointerException("Unchecked exception");
};
r.run();
```

Checked Exceptions (IOException, SQLException, etc.):

If the lambda expression throws a **checked exception**, the method that uses the lambda must declare that it throws the exception using the throws keyword, or the exception must be caught within the lambda.

This can become tricky because many functional interfaces in Java, such as Runnable or Consumer, don't declare any checked exceptions. To throw a checked exception in such cases, you must handle it with a try-catch block inside the lambda.

```java
Runnable r = () -> {
    try {
        throw new IOException("Checked exception");
    } catch (IOException e) {
        e.printStackTrace();
    }
};
r.run();
```

Alternatively, if you're using a method that allows checked exceptions, you can declare them as part of the method's throws clause
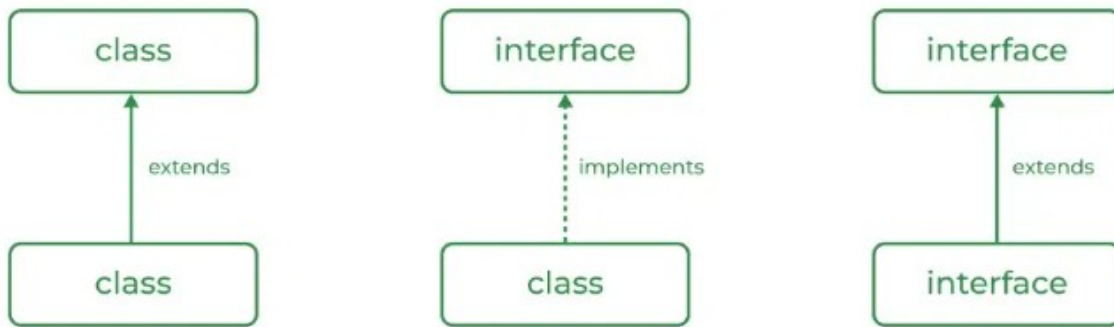
```java
Callable<Void> c = () -> {
    throw new IOException("Checked exception");
};
```

---

47. What Are the Rules We Need to Follow When Overriding a Method That Throws an Exception?

- **No new or broader checked exceptions**: The subclass cannot declare new or broader checked exceptions than the superclass method.

- **Unchecked exceptions**: The subclass can throw unchecked exceptions freely.
- **Narrower or no exceptions**: The subclass can declare fewer or more specific checked exceptions or none at all.

---

48. why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

---

49. Serializable Interface

- **Purpose**: Marks a class as capable of being serialized (converted into a byte stream), so it can be stored in files, transferred over a network, or persisted in databases.

- **Type**: Marker interface (no methods).

- **Use Case**: When you need to save the state of an object, you implement Serializable. The ObjectOutputStream and ObjectInputStream classes handle serialization and deserialization.

```java
import java.io.Serializable;

public class Example implements Serializable {
    private int id;
    private String name;
}
```

---

50. Cloneable Interface

- **Purpose**: Allows an object to be cloned (a duplicate copy of the object is created).

- **Type**: Marker interface.

- **Use Case**: Used when you want to create copies of an object. To make cloning possible, the Object.clone() method must be overridden.

```java
class Example implements Cloneable {
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

---

51. Comparable **Interface**

- **Purpose**: Defines a natural ordering for objects of a class. It contains a single method, compareTo(), which compares the current object with another object.

- **Type**: Functional interface.

- **Use Case**: Used to compare objects when sorting a collection like ArrayList or TreeSet.

```java
public class Example implements Comparable<Example> {
    private int id;

    @Override
    public int compareTo(Example other) {
        return this.id - other.id;
    }
}
```

---

52. Comparator **Interface**

- **Purpose**: Defines a custom order for objects of different classes. It has two methods: compare() and equals().

- **Use Case**: When you want to define multiple sorting orders for objects, Comparator is used. For example, you can sort by name, id, etc.

```java
import java.util.Comparator;

public class ExampleComparator implements Comparator<Example> {
    @Override
    public int compare(Example e1, Example e2) {
        return e1.getName().compareTo(e2.getName());
    }
}
```

---

53. Runnable **Interface**

- **Purpose**: Represents a task that can be executed by a thread. It contains a single method run(), which contains the code that defines the task.

- **Type**: Functional interface.

- **Use Case**: Used to implement multithreading. You pass a Runnable object to a Thread constructor.

```java
public class Task implements Runnable {
    @Override
    public void run() {
        System.out.println("Task is running");
    }
}

Thread thread = new Thread(new Task());
thread.start();
```

---

54. Iterable **Interface**

- **Purpose**: Represents a collection of objects that can be iterated (looped through). It provides the iterator() method, which returns an Iterator object.

- **Type**: Functional interface.

- **Use Case**: Allows collections like List, Set, etc., to be used in enhanced for loops (for-each)

```
Iterable<String> iterable = Arrays.asList("A", "B", "C");
for (String s : iterable) {
    System.out.println(s);
}
```

55. Types of Interfaces:

Normal Interface: This is a standard interface that declares methods without providing any implementations. The implementing class must provide implementations for all the methods.

Marker Interface: An interface with no methods or fields, used to signal or mark a class for a specific purpose. Common examples include Serializable, Cloneable, and Remote.

Functional Interface: An interface with exactly one abstract method. These interfaces can be used with lambda expressions. Java 8 introduced the @FunctionalInterface annotation to indicate that an interface is meant to be functional.

Tagging Interface: Similar to a marker interface, but it's used to tag or identify certain classes based on the implemented interface.

56. What is the difference between an interface and an abstract class in Java?

- An interface can have only abstract methods (before Java 8) and cannot contain method implementations (except for default and static methods in Java 8+).
- An abstract class can have both abstract and non-abstract methods.
- A class can implement multiple interfaces but can extend only one abstract class.

57. Can you declare a method in an interface as final or static?

You cannot declare a method in an interface as final, as methods in an interface are meant to be overridden by implementing classes. However, starting from Java 8, you can declare static methods in an interface.

58. What are the implications of making an interface method private in Java 9?

Java 9 introduced private methods in interfaces to allow code reusability within the interface itself. These private methods can only be used by other methods within the interface (like default or static methods) and are not accessible to implementing classes.

---

59. Diamond Problem and it's resolution:

When a class implements multiple interfaces having the same default method, ambiguity occurs because Java cannot decide which default implementation to use. To fix this, you override the method in your class and explicitly specify which interface's default method you want to call or give a custom logic by overriding it.

```java
interface Parent{  2 usages  3 implementations
    default void show(){  3 usages  3 overrides
        System.out.println("inside Parent");
    }
}
interface A extends Parent{  2 usages  1 implementation
    default void show(){  3 usages  1 override
        System.out.println("show A");
    }
}
interface B extends Parent{  2 usages  1 implementation
    default void show(){  3 usages  1 override
        System.out.println("show B");
    }
}
class C implements A,B{  2 usages
    @Override  3 usages
    public void show(){
        A.super.show();                 //we can use this
        B.super.show();                 //or this or both
        System.out.println("show C"); //or use custom logic
    }
}
class Test{
    public static void main(String []args){
        C c = new C();
        c.show();
```

---

60. How Can You Create a Thread Instance and Run It?

Creating a Thread by Extending the Thread Class

In this method, you create a subclass that extends Thread and overrides its run() method. The run() method contains the code that will execute in the new thread. To start the thread, you call the start() method, which internally invokes the run() method.

Creating a Thread by Implementing the Runnable Interface

In this approach, you create a class that implements the Runnable interface and pass an instance of this class to a Thread object. The Runnable interface requires implementing the run() method, where you define the thread's behavior.

```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code that will run in a new thread
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable();  // Create a Runnable instance
        Thread thread = new Thread(runnable);    // Pass it to a new Thread
        thread.start();  // Start the thread
    }
}
```

61. Describe the Different States of a Thread

New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

2. *Runnable State: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.*

A multi-threaded program allocates a fixed amount of time to each individual thread. Each and

every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

3. ***Blocked:*** *The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.*

4. ***Waiting state:*** *The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.*

5. ***Timed Waiting:*** *A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.*

6. ***Terminated State:*** *A thread terminates because of either of the following reasons:*

- *Because it exits normally. This happens when the code of the thread has been entirely executed by the program.*

- *Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.*

---

62. What is deadlock?

Deadlock occurs in a multithreaded environment when two or more threads are blocked forever, each waiting for the other to release a resource they need to proceed. It typically happens when threads acquire multiple locks and hold them while attempting to acquire additional locks held by other threads

---

63. Solid Principles:

Single Responsibility

Open/Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

Single Responsibility Principle: A class should have one, and only one, reason to change. This means that a class should only have one job or responsibility.

```
class Report {
    void generateReport() {
        // Logic to generate report
    }
}

class ReportPrinter {
    void printReport(Report report) {
        // Logic to print report
    }
}
```

Open-Closed Principle:

Open for Extension, Closed for Modification. classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code.

```
public class Guitar {

    private String make;
    private String model;
    private int volume;

    //Constructors, getters & setters
}
```

```
public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

Liskov Substitution Principle:

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. This principle ensures that a derived class can stand in for its base class.

```java
class Bird {
    void fly() {
        // Flying logic
    }
}

class Sparrow extends Bird {
    // Inherits fly() from Bird
}

class Ostrich extends Bird {
    void fly() {
        throw new UnsupportedOperationException("Ostriches can't fly!");
    }
}
```

Interface Segregation Principle:

larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

eg.

Imagine you are developing an application that sends messages via email. Initially, you may create a concrete EmailService class that your Notification class depends on directly. But later, if you need to send notifications via SMS or some other service, modifying the Notification class would violate the DIP

```java
// Abstraction (interface) that high-level module depends on
interface MessageService {
    void sendMessage(String message);
}

// Low-level concrete EmailService class implementing the abstraction
class EmailService implements MessageService {
    public void sendMessage(String message) {
        System.out.println("Email sent: " + message);
    }
}

// Low-level concrete SMSService class implementing the abstraction
class SMSService implements MessageService {
    public void sendMessage(String message) {
        System.out.println("SMS sent: " + message);
    }
}
```

64. What Is a Volatile keyword?

The volatile modifier in Java is used to indicate that a variable's value may be changed by different threads. When a variable is declared as volatile, the Java Memory Model guarantees that reads and writes to that variable are visible to all threads. This means that any thread that reads a volatile variable will see the most recently written value, preventing caching issues and ensuring visibility across threads.

65. What Is the Meaning of a Synchronized Keyword in the Definition of a Method? of a Static Method? Before a Block?

The synchronized keyword in Java is used to control access to a method or block of code by multiple threads. It ensures that only one thread can access the synchronized method or block at a time, preventing thread interference and maintaining data consistency.

When a method is declared with the synchronized keyword, it means that the method can only be accessed by one thread at a time for the given object. If one thread is executing a synchronized static method, no other thread can execute any synchronized static method on that class until the first thread exits the method.

A synchronized block is a more granular approach than synchronizing an entire method. It allows you to specify the object whose lock is to be acquired.

When a static method is declared as synchronized, it locks the class itself rather than a specific instance of the class.

---

67. If Two Threads Call a Synchronized Method on Different Object Instances Simultaneously, Could One of These Threads Block? What If the Method Is Static?

For synchronized instance methods, threads can run concurrently if they operate on different instances. They will not block each other.

For synchronized static methods, threads will block each other if they attempt to access the same static method on the same class. The lock is shared at the class level, so one thread will have to wait for the other to finish.

---

68. What Are the Benefits of Strings Being Immutable?

According to an interview by James Gosling, strings are immutable to improve performance and security.

And actually, we see several benefits to having immutable strings:

- **The string pool is only possible if the strings, once created, are never changed,** as they are supposed to be reused
- The code can **safely pass a string to another method**, knowing that it can't be altered by that method
- Immutably **automatically makes this class thread-safe**
- Since this class is thread-safe, **there is no need to synchronize common data**, which in turn improves performance
- Since they are guaranteed to not change, **their hashcode can be easily cached**

---

69. JVM Memory Management:

The JVM Memory Model defines how Java programs manage memory during execution. It consists of five main components:

1. **Method Area**: This stores class-level data like class metadata, static variables, and constants. It's shared among all threads.

2. **Heap**: The heap is where all objects and arrays are allocated. It's also shared among threads and is managed by the garbage collector. It's divided into the Young Generation (for new objects) and Old Generation (for long-lived objects).

3. **Stack**: Each thread has its own stack that stores method call information, local variables, and partial results. Each method invocation creates a stack frame, which is destroyed when the method completes.

4. **Program Counter (PC) Register**: Every thread has its own PC register, which keeps track of the current instruction being executed.

5. **Native Method Stack**: This is used when executing native methods through the Java Native Interface (JNI).

Memory management is handled by the garbage collector, which automatically frees memory occupied by unreachable objects in the heap.

---

70. How Is a String Stored in Memory?

String is an object stored in the Heap memory, but because it's immutable, the JVM maintains a special area called the String Pool inside the heap. When a string is created using a string literal, the JVM first checks the String Pool to see if an identical string already exists. If it does, the reference to the existing string is returned. If not, a new string is created in the pool.

When you create a string using the new keyword, it directly creates a new object in the heap, bypassing the String Pool.

The immutability of strings ensures that once a string is created, it can't be modified, which is why it can be safely shared among multiple parts of an application.

---

71. Are Interned Strings Eligible for Garbage Collection in Java?

starting from Java 7, interned strings can be garbage-collected when they are no longer referenced.

---

72. What Is the String Constant Pool?

The string pool, also known as the String constant pool or the String intern pool, is a special memory region where the JVM stores String instances.

---

73. Is String Thread-Safe? How?

Strings are indeed completely thread-safe because they are immutable. Any class which is immutable automatically qualifies for thread-safety because its immutability guarantees that its instances won't be changed across multiple threads.

---

74. What's the Difference Between str1 == str2 and str1.equals(str2)?

== compares references (checks if both strings point to the same object).
equals() compares the actual content (checks if the character sequences are identical).

---

75. How Can We Split a String in Java?

You can split a string in Java using the split() method from the String class, providing a regular expression to define the delimiter. You can also specify a limit to control the number of resulting substrings.

---

76. What Is Stringjoiner?

StringJoiner is a class introduced in Java 8 for joining separate strings into one, like taking a list of colors and returning them as a comma-delimited string. We can supply a delimiter as well as a prefix and suffix:

```java
StringJoiner joiner = new StringJoiner(",", "[", "]");
joiner.add("Red")
   .add("Green")
   .add("Blue");

assertEquals("[Red,Green,Blue]", joiner.toString());
```

---

77. Difference Between String, Stringbuffer and Stringbuilder?

Strings are immutable. This means that if we try to change or alter its values, then Java creates an absolutely new String.

Both StringBuilder and StringBuffer in Java create objects that hold a mutable sequence of characters. StringBuffer is synchronized and therefore thread-safe whereas StringBuilder is not.

---

78. What Does String's intern() Method Do?

The intern() method checks the String Pool, a special memory area in the heap where Java stores unique string literals. If the string already exists in the pool, it returns a reference to that string; if not, it adds the string to the pool and then returns its reference.

```java
String str1 = new String("hello");
String str2 = "hello";
String str3 = str1.intern();

System.out.println(str1 == str2); // false, different references
System.out.println(str2 == str3); // true, both refer to the same interned string
```

79. What Is String.format() and How Can We Use It

String.format() is a method in Java that allows for the creation of formatted strings by using format specifiers to define how values should be formatted and inserted into a string. It provides more flexibility and readability compared to simple string concatenation

```java
String name = "Alice";
int age = 30;
double salary = 12345.67;

// Using String.format() to create a formatted string
String formattedString = String.format("Name: %s, Age: %d, Salary: %.2f", name, age, salary

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

Output:

```yaml
Name: Alice, Age: 30, Salary: 12345.67
```

80. How Would We Convert a Java String into a Byte Array?

To convert a Java string into a byte array, you can use the getBytes() method of the String class. You can either use the default charset or specify one, such as UTF-8, to ensure consistent encoding. This conversion is often necessary for tasks like file I/O or network communication.

```java
String originalString = "Hello, World!";
byte[] byteArray = originalString.getBytes(); // Using default charset


System.out.println(Arrays.toString(byteArray));
```

```java
import java.nio.charset.StandardCharsets;
import java.util.Arrays;


String originalString = "Hello, World!";
byte[] byteArray = originalString.getBytes(StandardCharsets.UTF_8); // Specifying UTF-8


System.out.println(Arrays.toString(byteArray));
```

---

81. How Can You Remove Whitespace from a String in Java?

trim(): Removes leading and trailing whitespace.

replaceAll("\\s", ""): Removes all whitespace, including spaces between words.

\\s is a regular expression that matches any whitespace character (spaces, tabs, newlines).

```java
Collection<String> myCollection = new ArrayList<>();
myCollection.add("Hello");
myCollection.add("World");
```

```java
List<String> myList = new ArrayList<>();
myList.add("Apple");
myList.add("Banana");
Collections.sort(myList);  // Sorts the list
System.out.println(myList);  // Output: [Apple, Banana]
```

---

82. Collection vs Collections:

Collection: A core interface of the Java Collections Framework, representing a group of objects.

Collections: A utility class that provides static methods to perform operations on collections like sorting, searching, and making them thread-safe.

83. iterator vs enumeration:

Enumeration is a legacy interface that is primarily used for read-only traversal of older collections like Vector and Hashtable.

Iterator is the modern and more versatile interface, providing additional functionality like element removal during iteration. It is widely used in the Java Collections Framework. For most modern Java development, Iterator is preferred over Enumeration.

---

84. What Is Garbage Collection and What Are Its Advantages?

Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

---

85. Are There Any Disadvantages of Garbage Collection?

Yes. Whenever the garbage collector runs, it has an effect on the application's performance. This is because all other threads in the application have to be stopped to allow the garbage collector thread to effectively do its work.

---

86. What Is Generational Garbage Collection?

Generational Garbage Collection is a memory management strategy used by the Java Virtual Machine (JVM) to optimize garbage collection by categorizing objects based on their lifespan.

Young Generation:

- Newly created objects are allocated here.
- Objects that survive multiple garbage collection cycles in the young generation are moved to the **Old Generation**.
- **Minor GC**: Garbage collection in the young generation is called a **Minor GC**, and it occurs frequently but is fast since most objects here are short-lived and can be quickly reclaimed.

Old Generation (Tenured Generation)

- Objects that have survived several cycles in the young generation are promoted to the old generation.

- **Major GC**: Garbage collection in the old generation is called a **Major GC** or **Full GC**. It occurs less frequently but takes longer as it deals with larger and more complex objects.

87. What Happens When There Is Not Enough Heap Space to Accommodate Storage of New Objects?

If there is no memory space for creating a new object in Heap, Java Virtual Machine throws OutOfMemoryError or more specifically java.lang.OutOfMemoryError heap space.

---

88. Describe Strong, Weak, Soft and Phantom References

- **Strong Reference:** The default reference; objects won't be collected as long as a strong reference exists.

```
String strongRef = new String("Hello");
```

- **Weak Reference**: Objects are collected if no strong references exist. Useful for memory-sensitive data structures.

```
WeakReference<String> weakRef = new WeakReference<>(new String("Weak Hello"));
```

- **Soft Reference**: Same as weak only diff. Is objects are collected only when the JVM is low on memory, useful for caching.

```
SoftReference<String> softRef = new SoftReference<>(new String("Soft Hello"));
```

- **Phantom Reference**: Used to determine when an object has been finalized and is ready for garbage collection, useful for resource cleanup and post-GC actions.

    eg. PhantomReference<String> phantomRef = new PhantomReference<>(new String("Phantom Hello"), referenceQueue);

---

89. Suppose We Have a Circular Reference (Two Objects That Reference Each Other). Could Such Pair of Objects Become Eligible for Garbage Collection and Why?

Yes, a circular reference (two or more objects referencing each other) can still become eligible for garbage collection in Java. The garbage collector works by tracing reachable objects from root references (e.g., from the main thread, static variables, local variables on the stack). If an object or

group of objects (such as a circular reference) is not reachable from any root reference, it will be considered garbage, regardless of internal references between those objects.

---

90. How Are Strings Represented in Memory?

String is an object that consists of two main components:

1. **byte[] value**: This is an array of characters that stores the actual string.
2. **int hash**: This field holds the hash code of the string, which is calculated the first time you call hashCode() and is cached for future use. If the hash code is zero, it will be recalculated each time hashCode() is called.

Before java 9 char[] was used

---

91. 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.

---

92. 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.

---

93. What is Dependency Injection?

The most common way IoC(Inversion Of Control) is implemented in Spring. DI is a design pattern used to implement IoC, where the Spring container injects the dependencies into a class at runtime rather than at compile time

---

94. Types of Dependency Injection:

a. Constructor Injection: Dependencies are provided through a class constructor.

b. Setter Injection: Dependencies are provided through setter methods.

c. Field Injection: Dependencies are directly assigned to fields. (Less preferred due to testing and maintenance difficulties.)

95. Which Is the Best Way of Injecting Beans and Why?

The recommended approach is to use constructor arguments for mandatory dependencies and setters for optional ones. This is because constructor injection allows injecting values to immutable fields and makes testing easier.

---

96. BeanFactory vs ApplicationContext:

In the Spring Framework, BeanFactory and ApplicationContext are both used for managing beans, but they serve different purposes. BeanFactory is a basic container that supports Dependency Injection with lazy initialization, meaning beans are created only when needed, making it suitable for lightweight applications. On the other hand, ApplicationContext is an advanced container that extends BeanFactory and provides additional features like eager initialization(creating and initializing beans at the startup of the application, rather than waiting until the beans are first requested) of singleton beans, internationalization, event propagation, and support for Aspect-Oriented Programming (AOP). This makes ApplicationContext the preferred choice for most enterprise applications. While BeanFactory has a lower memory footprint, ApplicationContext enhances overall application performance and offers more capabilities, making it ideal for complex use cases.

---

97. What Is a Spring Bean?

The Spring Beans are Java Objects that are initialized by the Spring IoC container.

---

98. What Is the Default Bean Scope in Spring Framework?

By default, a Spring Bean is initialized as a singleton.

---

99. How to Define the Scope of a Bean?

In order to set Spring Bean's scope, we can use @Scope annotation or "scope" attribute in XML configuration files. Note that there are five supported scopes:

Singleton

Prototype

Request

Session

Global-session

Singleton: Single instance throughout the application.

Prototype: A new instance is created for each request.

Request: A new instance is created for each HTTP request.

Session: A new instance is created for each user session.

Global-session: scope is used in **portlet-based web applications**. It ties a bean to a global session, which is shared across multiple portlets for the same user within a portal. Unlike the regular session scope (limited to one portlet), the global session is accessible across all portlets

---

100. Are Singleton Beans Thread-Safe?

No, singleton beans are not thread-safe, as thread safety is about execution, whereas the singleton is a design pattern focusing on creation. Thread safety depends only on the bean implementation itself.

---

101. Spring Security:

Spring Security is a robust framework that provides a comprehensive security solution for Java applications, enabling authentication, authorization, and protection against various security threats.

---

102. Design Patterns:

Singleton Pattern

- **Definition**: Ensures that a class has only one instance and provides a global point of access to it.
- **Usage in Spring**: By default, Spring beans are singletons. This means that the Spring container creates a single instance of a bean and returns that instance whenever it is requested.
- **Benefits**: Saves memory and resources by reusing the same instance and provides controlled access to shared resources.

Factory Pattern

- **Definition**: A creational pattern that provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created.
- **Usage in Spring**: Spring uses the Factory pattern through the ApplicationContext, which creates beans based on the configuration provided (XML, Java annotations, or Java configuration).

- **Benefits**: Promotes loose coupling by separating the instantiation of objects from their usage.

---

103. How Does the Scope Prototype Work?

Scope prototype means that every time we call for an instance of the Bean, Spring will create a new instance and return it. This differs from the default singleton scope, where a single object instance is instantiated once per Spring IoC container.

---

104. What Is Spring JdbcTemplate Class

JdbcTemplate is a core class in the Spring framework that simplifies database interactions in Java applications by providing a higher-level abstraction over JDBC. It reduces boilerplate code required for executing SQL queries, managing connections, and handling exceptions. With automatic connection management, it ensures that database connections are properly opened and closed. It also translates SQL exceptions into Spring's DataAccessException, making error handling more straightforward. Additionally, JdbcTemplate supports object mapping through methods like query(), which retrieves a list of objects, and update(), which executes SQL updates. It allows the use of prepared statements to prevent SQL injection and improve performance. Overall, JdbcTemplate streamlines the process of working with databases, making it an essential tool for Spring developers.

---

105. What is Spring DAO?

Spring DAO (Data Access Object) is a design pattern used in the Spring Framework to simplify how applications interact with data sources, like databases. It helps separate the data access logic from the main business logic of the application, making the code easier to manage and understand. With Spring DAO, you have a consistent way to perform data operations, which means you can easily switch between different types of databases or data sources without changing much of your code. It also provides a uniform way to handle exceptions, turning complicated database errors into simpler ones that are easier to deal with. Additionally, Spring DAO integrates well with other Spring features, such as transaction management, allowing you to handle transactions smoothly. Overall, Spring DAO helps developers write cleaner, more organized code when accessing and manipulating data.

---

106. Aspect-Oriented Programming (AOP) is a programming technique used to modularize cross-cutting concerns, like logging, security, and transaction management, that affect multiple parts of an application. Instead of scattering these concerns across various methods, AOP isolates them into a separate module called an **aspect**.
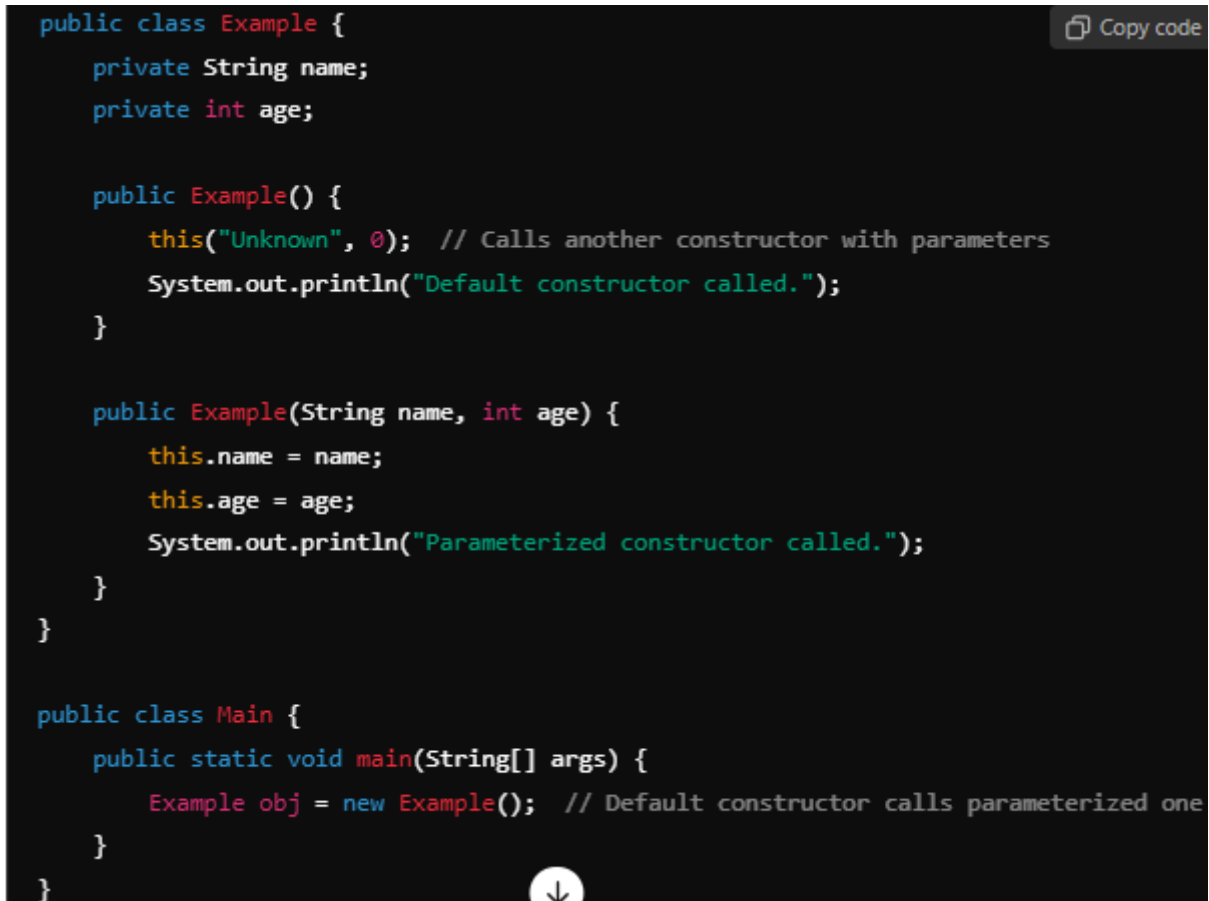
Key Concepts:

- **Aspect**: A module that encapsulates a cross-cutting concern.
- **Advice**: The action to be taken by the aspect (e.g., before or after a method).
- **Join Point**: A point in the program execution (e.g., method execution) where an aspect can be applied.
- **Pointcut**: Defines where an aspect should be applied (which methods or classes).
- **Weaving**: The process of applying aspects to the target object, either at compile-time, load-time, or runtime.

---

107. What Is Constructor Chaining?

Constructor Chaining is the process of calling one constructor from another constructor in the same class or a superclass, to initialize an object with multiple constructors in a structured way. This helps in reusing code and avoiding redundancy by having common initialization logic in a single constructor.

```java
public class Example {
    private String name;
    private int age;

    public Example() {
        this("Unknown", 0);  // Calls another constructor with parameters
        System.out.println("Default constructor called.");
    }

    public Example(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Parameterized constructor called.");
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj = new Example();  // Default constructor calls parameterized one
    }
}
```

```java
class Parent {
    public Parent() {
        System.out.println("Parent constructor called.");
    }
}

class Child extends Parent {
    public Child() {
        super();  // Calls the Parent class constructor
        System.out.println("Child constructor called.");
    }
}

public class Main {
    public static void main(String[] args) {
        Child childObj = new Child();  // Both constructors are called
    }
}
```

---

108. Static Nested Class:

A static nested class is a class that is defined within another class, and it is marked with the static keyword. Unlike regular inner classes, a static nested class does not have a reference to the instance of the outer class. Therefore, it cannot access non-static members of the outer class directly.

---

109. What is Auto boxing and Auto unboxing?

Auto-boxing
Auto-boxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. This feature was introduced in Java 5.

Example:
```java
int primitiveInt = 5;
Integer boxedInt = primitiveInt; // Auto-boxing
```
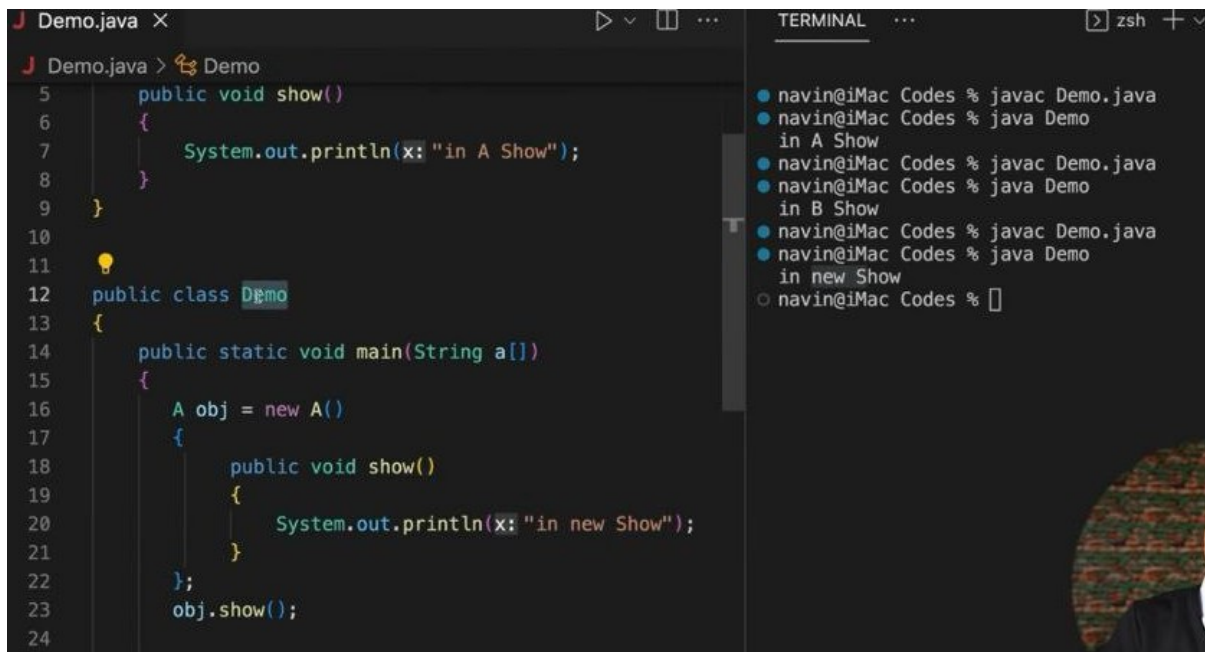
Auto-unboxing
Auto-unboxing is the automatic conversion that the Java compiler makes from the wrapper class to the primitive type.

Example:

```java
Integer boxedInt = new Integer(5);
int primitiveInt = boxedInt; // Auto-unboxing
```

---

110. What Is an Anonymous Class?

Anonymous class is a one-shot class that is defined in the same place where its instance is needed. This class is defined and instantiated in the same place, thus it does not need a name.

---

101. Explain Autowiring and its types.

Autowiring reduces the efforts of object instantiation by auto injection of dependencies into beans managed by spring.

@Autowired: Most commonly used annotation for dependency injection by type.

byType: Injects dependencies based on the type of the bean.
byName: Matches the bean name with the property name for injection.
constructor: Injects dependencies via constructor matching.

---

102. Explain Spring Boot and its advantages.

Spring Boot framework is built to simplify the process of development and deployment for Spring applications. It reduces the configuration efforts by auto-configuring beans and contiainers making the development faster and more efficient.
Advantages of Spring Boot:

•**Reduced configuration:** Reduces XML configuration files, making development configuration easy.

•**Automatic configuration:** Automatically configures beans and management of dependencies.

•**Embedded server:** *Tomcat* is a built-in server that reduces the effort for separate server configuration.

•**Starter POMs:** Provides pre-configured starter POMs that simplify dependency management.

•**Rapid application development:** Enables rapid application development by simplifying the development process and reducing configuration overhead.

•**Cloud-friendly:** Best choice for cloud deployment with support across various cloud platforms.

---

103. Explain some of the most used Spring Boot annotations

@SpringBootApplication:

Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. It marks the main class for the Spring Boot application.

@RestController:

A specialized @Controller for creating RESTful web services. Automatically serializes responses to JSON or XML.

@RequestMapping:

Maps HTTP requests to specific controller methods. Can define HTTP method, URL path, and parameters.

@Autowired:

Enables automatic dependency injection, allowing Spring to resolve and inject collaborating beans.

@Service:

Indicates that a class contains business logic and is part of the service layer.

@Repository:

Marks a class as a Data Access Object (DAO) for database interactions, encapsulating CRUD operations.

@Component:

A generic stereotype for any Spring-managed component, used for automatic bean detection.

@Configuration:

Indicates that a class contains one or more @Bean methods for defining configuration.

@Value:

Injects external values into fields from properties files (e.g., application.properties).

@Profile:

Indicates that a bean is eligible for registration when a specified profile is active, enabling environment-specific configurations.

---

104. Explain the internal working of *@SpringBootApplication*

The *@SpringBootApplication* annotation performs several tasks as it contains features of three annotation will are *@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan.*
•*@SpringBootApplication* enables component scanning for beans and auto-configuration.

•Registers various beans, including Spring Boot auto-configuration beans.

•Instantiates the embedded server

•Runs the application

---

105. What are Profiles in Spring Boot

In Spring Boot, Profiles allow configuration for applications differently in different environments, such as
•Development

•Staging

•Production

Separate configuration files are defined for each profile, which can be activated using environment variables or command-line arguments.

---

106. Explain Spring JDBC API and its classes.

•**Spring JDBC:** Spring provides a simple way in the form of a JDBC abstraction layer to establish a bridge between database and application. It reduces boilerplate code and configurations.

•**Key classes:**

JdbcTemplate: Provides simple methods for executing SQL statements and working with data exchange for applications.

DataSource: Establish the connection(bridge) of data exchange from database.

SimpleJdbcCall: method present in Spring JDBC API, used for interacting with database-stored procedures.

---

107. Explain Spring Data and What is Data JPA?

Spring Data is a powerful framework that can be used to develop data-oriented applications. It aims to simplify the development of data-centric applications by offering abstractions, utilities, and integration with various data sources.
•**Spring Data JPA:** This project provides support for accessing data from relational databases using JPA.

---

108. Mention the steps to connect the Spring Boot application to a database using JDBC.

To connect an external database like MySQL or Oracle to a Spring Boot application using JDBC, we need to follow below steps:
•Add the dependency for the JDBC driver of the database.

•Create an application.properties file.

•Configure the database connection properties.

•Create a JdbcTemplate bean.

•Use the JdbcTemplate bean to execute SQL queries and statements.

---

109. Method Reference:

Method reference is a feature in Java that allows you to refer to methods by their names, which can be used as a shorthand to call methods without needing to explicitly invoke them. It provides a clear and concise way to use existing methods as lambda expressions, enhancing the readability of your code.

Static Method Reference:

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Using method reference to a static method
names.forEach(System.out::println);
```

Instance Method Reference :

```java
public class MethodReferenceExample {
    public void printName(String name) {
        System.out.println(name);
    }

    public static void main(String[] args) {
        MethodReferenceExample example = new MethodReferenceExample();
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using method reference to an instance method of a specific object
        names.forEach(example::printName);
    }
}
```

Instance Method Reference of an Arbitrary Object of a Particular Type:

```java
import java.util.Arrays;
import java.util.List;

public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using method reference to an instance method of an arbitrary object
        names.sort(String::compareToIgnoreCase);
        System.out.println(names);
    }
}
```

Constructor Reference:

```java
import java.util.ArrayList;
import java.util.List;

public class MethodReferenceExample {
    public static void main(String[] args) {
        // Using method reference to a constructor
        List<String> names = createList(ArrayList::new);
        names.add("Alice");
        names.add("Bob");
        System.out.println(names);
    }

    public static <T> List<T> createList(Supplier<List<T>> supplier) {
        return supplier.get();
    }
}
```

When Lambda Expressions Can Be Replaced by Method References

Single Abstract Method: A lambda can be replaced by a method reference if it matches the signature of a method in a functional interface with a single abstract method (SAM).

```java
// Using a lambda expression
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));

// Replaced with a method reference
names.forEach(System.out::println);
```

Directly Calling an Existing Method: If the lambda expression simply calls an existing static method, instance method, or constructor without any additional logic, it can be replaced by a method reference.

```java
// Lambda expression
Comparator<String> comparator = (s1, s2) -> s1.compareToIgnoreCase(s2);

// Replaced with a method reference
Comparator<String> comparator = String::compareToIgnoreCase;
```

When Lambda Expressions Cannot Be Replaced by Method References:

Complex Logic: If the lambda expression contains more than just a method call, such as additional logic, conditionals, or multiple statements, it cannot be replaced by a method reference.

```java
// Lambda expression with additional logic
names.forEach(name -> {
    if (name.startsWith("A")) {
        System.out.println(name);
    }
});
```

Multiple Arguments with Logic: If the lambda performs some operations or calculations based on its parameters, a method reference cannot replace it.

```java
// Lambda with multiple arguments and logic
BiFunction<Integer, Integer, Integer> adder = (a, b) -> {
    if (a < 0 || b < 0) return 0;
    return a + b;
};
```

Non-Matching Method Signatures: If there is no existing method with a matching signature in a compatible functional interface, a method reference cannot be used.

```java
// Lambda expression with a non-matching method signature
names.forEach(name -> {
    System.out.println(name.length());
});
```

---

110. Streams:

streams are a powerful and modern API introduced in Java 8 to process collections of data in a declarative way. They allow you to perform operations like filtering, mapping, and reducing on data in a clean and efficient manner, often using functional programming concepts like lambda expressions and method references.

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // Stream pipeline example
        List<String> filteredNames = names.stream()          // Source: Crea
                .filter(name -> name.startsWith("A"))         // Intermediate
                .map(String::toUpperCase)                     // Intermediate
                .collect(Collectors.toList());                // Terminal ope

        System.out.println(filteredNames);  // Output: [ALICE]
    }
}
```

Operations:

Intermediate Operations:

filter(Predicate<T> predicate): Filters elements based on a condition.

List<String> filtered = list.stream().filter(s -> s.length() > 3).collect(Collectors.toList());

map(Function<T, R> mapper): Transforms each element by applying a function.

List<Integer> lengths = list.stream().map(String::length).collect(Collectors.toList());

sorted(Comparator<T> comparator): Sorts the elements.

List<String> sorted = list.stream().sorted().collect(Collectors.toList());

distinct(): Removes duplicates from the stream.

List<String> distinctNames = list.stream().distinct().collect(Collectors.toList());

Terminal Operations:

collect(Collector<T, A, R> collector): Collects the result of the stream into a collection or other data structure.

List<String> collected = list.stream().collect(Collectors.toList());

forEach(Consumer<T> action): Iterates over each element and performs the given action.

list.stream().forEach(System.out::println);

reduce(BinaryOperator<T> accumulator): Combines all elements into a single result.

int sum = numbers.stream().reduce(0, Integer::sum);

count(): Returns the number of elements in the stream.

long count = list.stream().count();

findFirst() / findAny(): Returns the first element (or any element in parallel streams) wrapped in an Optional.

Optional<String> first = list.stream().findFirst();

Optional<String> anyName = names.stream().findAny();

flatMap(Function<T, Stream<R>> fun):

Returns Stream<R>, which is a flattened stream of the results obtained by applying the function to each element of the original stream. It used to transform a stream of elements into a stream of streams, then flatten it into a single stream. It is particularly useful when dealing with collections of collections, such as lists of lists, and you want to process all the nested elements as a single, unified stream.

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class FlatMapExample {
    public static void main(String[] args) {
        List<List<String>> listOfLists = Arrays.asList(
            Arrays.asList("Alice", "Bob"),
            Arrays.asList("Charlie", "David"),
            Arrays.asList("Eve", "Frank")
        );

        // Using flatMap to flatten and process the nested lists
        List<String> flatList = listOfLists.stream()
                                            .flatMap(List::stream)   // Flattening
                                            .collect(Collectors.toList());

        System.out.println(flatList);
    }
}
```

Parallel Streams

Java streams can also be **parallelized** for better performance when working with large datasets. Parallel streams divide the work across multiple threads.

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

// Parallel stream example
List<String> filteredNames = names.parallelStream()
        .filter(name -> name.startsWith("A"))
        .map(String::toUpperCase)
        .collect(Collectors.toList());
```

---

111. Why streams are lazy in nature?

Streams are lazy because they delay the execution of operations until a final result is needed. This means that intermediate actions like filtering or mapping data don't run immediately. Instead, they wait for a terminal operation, like collecting or printing the results, to trigger them. This approach makes streams more efficient, as they only process the data that's actually required. For example,

streams can stop processing early if an operation like `findFirst()` is called, saving time and resources. By processing data one item at a time and combining multiple steps into a single pass, streams handle large or complex datasets efficiently without using too much memory.

---

112. Sort Stream:

ASC:

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamSortExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);

        // Sorting the stream of integers
        List<Integer> sortedNumbers = numbers.stream()   // Convert list to stream
                .sorted()                                 // Sort the stream in natural o
                .collect(Collectors.toList());            // Collect the sorted elements

        System.out.println("Sorted Numbers: " + sortedNumbers);
    }
}
```

DESC:

```java
import java.util.Comparator;

public class StreamSortExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);

        // Sorting the stream of integers in descending order
        List<Integer> sortedNumbersDesc = numbers.stream()
                .sorted(Comparator.reverseOrder())       // Sort in descending orde
                .collect(Collectors.toList());

        System.out.println("Sorted Numbers (Descending): " + sortedNumbersDesc);
    }
}
```

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamSortExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);

        // Sorting in descending order using a lambda expression
        List<Integer> sortedNumbersDesc = numbers.stream()
                .sorted((a, b) -> b.compareTo(a))    // Custom comparator to rever
                .collect(Collectors.toList());

        System.out.println("Sorted Numbers (Descending): " + sortedNumbersDesc);
    }
}
```

---

113. Reflection API:

The Reflection API in Java allows you to inspect and manipulate the runtime behavior of classes, methods, fields, and constructors, even if they are private or protected. It enables dynamic code execution, which can be very powerful, but it should be used with caution because it can break encapsulation, degrade performance, and increase the complexity of your code.

Key Classes in Java Reflection API
- **Class**: Represents the classes and interfaces in a running Java application. It allows you to get metadata about a class.
- **Field**: Allows you to inspect and modify fields (variables) of a class.
- **Method**: Allows you to inspect and invoke methods of a class.
- **Constructor**: Allows you to inspect and create new instances of a class.

Commonly Used Reflection Methods

1. **Class.forName(String className)**: Loads the class dynamically at runtime.
2. **Class.getDeclaredFields()**: Retrieves all fields (including private fields) of the class.
3. **Class.getDeclaredMethods()**: Retrieves all methods (including private methods) of the class.
4. **Class.getDeclaredConstructors()**: Retrieves all constructors (including private constructors) of the class.
5. **Field.setAccessible(true)**: Allows you to modify private fields.
6. **Method.invoke(Object obj, Object... args)**: Invokes a method on a specific object instance dynamically.
7. **Constructor.newInstance(Object... initargs)**: Creates a new instance of the class using the specified constructor.

Key Use Cases for Reflection

1. **Frameworks**: Reflection is heavily used in frameworks like Spring and Hibernate to dynamically inspect and modify objects.
2. **Testing**: Allows unit testing frameworks (like JUnit) to invoke private methods or access private fields.
3. **Serialization**: Reflection is used in serialization frameworks to inspect and serialize objects.
4. **Dynamic Class Loading**: Java reflection enables loading and using classes at runtime that are unknown at compile-time.

---

114. Sealed Class:

Sealed class is a class that restricts which other classes or interfaces can extend or implement it. This feature was introduced in Java 15 as a preview feature and finalized in Java 17.

```
public sealed class Shape permits Circle, Rectangle, Square {
    // class body
}

public final class Circle extends Shape {
    // class body
}

public sealed class Rectangle extends Shape permits ColoredRectangle {
    // class body
}

public non-sealed class Square extends Shape {
    // class body
}

public final class ColoredRectangle extends Rectangle {
    // class body
}
```

---

115. Java 8 interfaces:

Here's an updated, interview-friendly explanation of all key functional interfaces in Java — including:

1. **Simple example (Example 1)**

2. **Stream use case (Example 2)**

This will help you understand how these are used in real-time scenarios like filtering, mapping, printing, etc.

✅ **1. Predicate**

Checks a condition — returns true or false.

🔷 **Example 1:**

Predicate<Integer> isEven = num -> num % 2 == 0;
System.out.println(isEven.test(4)); // true

🔷 Example 2 (Stream):

```
List<Integer> nums = List.of(1, 2, 3, 4, 5);
nums.stream()
    .filter(isEven)
    .forEach(System.out::println); // 2, 4
```

✅ 2. Function<T, R>

Takes one input and returns a result.

🔷 Example 1:

```
Function<String, Integer> strLength = str -> str.length();
System.out.println(strLength.apply("Java")); // 4
```

🔷 Example 2 (Stream):

```
List<String> names = List.of("Tom", "Jerry");
names.stream()
    .map(strLength)
    .forEach(System.out::println); // 3, 5
```

✅ 3. Consumer

Consumes input — returns nothing.

🔷 Example 1:

```
Consumer<String> greet = name -> System.out.println("Hello " + name);
greet.accept("Alice"); // Hello Alice
```

🔷 Example 2 (Stream):

```
List<String> names = List.of("Ram", "Shyam");
names.stream()
    .forEach(greet); // Hello Ram, Hello Shyam
```

✅ 4. Supplier

Supplies a value — takes no input.

🔷 Example 1:

```
Supplier<String> msg = () -> "Hello, Supplier!";
```

```
System.out.println(msg.get()); // Hello, Supplier!
```

🔷 **Example 2 (Stream usage — not direct, but can generate):**

```
Stream<String> s = Stream.generate(msg).limit(2);
s.forEach(System.out::println);
// Hello, Supplier!
// Hello, Supplier!
```

✅ **5. BiPredicate<T, U>**

Takes 2 inputs — returns boolean.

🔷 **Example 1:**

```
BiPredicate<String, String> isMatch = (a, b) -> a.equalsIgnoreCase(b);
System.out.println(isMatch.test("Java", "java")); // true
```

🔷 **Example 2 (Stream):**

```
List<String> list = List.of("java", "Python", "c++");
String key = "JAVA";
list.stream()
    .filter(item -> isMatch.test(item, key))
    .forEach(System.out::println); // java
```

✅ **6. BiFunction<T, U, R>**

Takes 2 inputs — returns a result.

🔷 **Example 1:**

```
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
System.out.println(add.apply(2, 3)); // 5
```

🔷 **Example 2 (Stream):**

```
List<Integer> list1 = List.of(1, 2, 3);
List<Integer> list2 = List.of(4, 5, 6);

IntStream.range(0, list1.size())
    .mapToObj(i -> add.apply(list1.get(i), list2.get(i)))
    .forEach(System.out::println); // 5, 7, 9
```

✅ 7. BiConsumer<T, U>

Takes 2 inputs — returns nothing.

🔷 Example 1:

```
BiConsumer<String, Integer> printAge = (name, age) ->
   System.out.println(name + " is " + age + " years old");
printAge.accept("Tom", 25); // Tom is 25 years old
```

🔷 Example 2 (Stream):

```
Map<String, Integer> map = Map.of("Alice", 22, "Bob", 30);
map.forEach(printAge);
// Alice is 22 years old
// Bob is 30 years old
```

🚫 8. BiSupplier

❌ Not available in standard Java.

✅ 9. UnaryOperator

Special case of Function<T, T> — same input & output type.

🔷 Example 1:

```
UnaryOperator<Integer> square = x -> x * x;
System.out.println(square.apply(5)); // 25
```

🔷 Example 2 (Stream):

```
List<Integer> numbers = List.of(2, 3, 4);
numbers.stream()
    .map(square)
    .forEach(System.out::println); // 4, 9, 16
```

✅ 10. BinaryOperator

Special case of BiFunction<T, T, T> — 2 same inputs & 1 same output type.

🔷 Example 1:

```
BinaryOperator<Integer> multiply = (a, b) -> a * b;
```

System.out.println(multiply.apply(3, 4)); // 12

◆ **Example 2 (Stream reduce):**

List<Integer> nums = List.of(1, 2, 3, 4);
Integer product = nums.stream()
            .reduce(1, multiply);
System.out.println(product); // 24

116.

Ternary Operator (? :)

The ternary operator is a **shortcut for an if-else statement**. It takes **three parts**:

- A condition

- A value if the condition is true

- A value if the condition is false

eg. condition ? valueIfTrue : valueIfFalse;

String result = (age >= 18) ? "Adult" : "Minor";

117.

Unary Operator

A unary operator works on **only one operand** (one value). It usually does things like:

- Increment or decrement (++, --)

- Negate a value (-)

- Logical NOT (!)

---

118. Ambiguity problem:

Ambiguity happens when the Java compiler cannot decide which method to call because two or more methods look equally good (ambiguous) to match a method call. This usually occurs in method overloading or multiple inheritance (interfaces) situations.

 Common scenarios causing ambiguity:

- Overloaded methods with parameters that can match the call in multiple ways (e.g., int vs long vs Integer).

- A class implementing two interfaces that have methods with the same signature and default implementations. (Diamond problem)
- Varargs vs fixed-parameter overloading.

---

119. Why default implementation of method of interface's was introduced?

Before Java 8, interfaces could only have abstract methods — meaning all methods had to be implemented by the class. But this caused a big problem when Java wanted to add new methods to existing interfaces like List, Map, or Iterable.

If they added a new method, then every class that used those interfaces would break — because they would be forced to implement the new method.

---

120. How to create thread safe singleton class in Java?

```java
public class MySingleton {

    private static volatile MySingleton instance;

    // Private constructor to prevent object creation
    private MySingleton() {}

    public static MySingleton getInstance() {
        if (instance == null) {
            synchronized (MySingleton.class) {
                if (instance == null) {
                    instance = new MySingleton(); // Only one thread will create the instance
                }
            }
        }
        return instance;
    }
}
```

Instead of synchronized block we use synchronized to getInstance() method?

You can make the getInstance() method synchronized so only one thread can call it at a time. Every call to getInstance() is synchronized, so even after the instance is created, threads wait unnecessarily. This can slow down your app if getInstance() is called very frequently.

121. When can't we use synchronized(this)?

You cannot use synchronized(this) if:

1. You are inside a static method
   Because static methods belong to the class, not to any specific object. So there is no this in static methods.

2. You want to synchronize across all instances
   synchronized(this) locks only the current object instance. If you want to lock for the whole class, so no two instances interfere, you need to lock on the class object like synchronized(MyClass.class).