

JAVA

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

```
public class Player {  
    private int ranking;  
    private String name;  
    private int age;  
  
    // constructor, getters, setters  
}
```

PlayerSorter.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:

```
public class Player implements Comparable<Player> {  
  
    // same as before  
  
    @Override  
    public int compareTo(Player otherPlayer) {  
        return Integer.compare(getRanking(), otherPlayer.getRanking());  
    }  
  
}
```

Using comparator:

```
public class PlayerRankingComparator implements Comparator<Player> {  
  
    @Override  
    public int compare(Player firstPlayer, Player secondPlayer) {  
        return Integer.compare(firstPlayer.getRanking(), secondPlayer.getRanking());  
    }  
  
}
```

```
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 <i>java.lang</i> package since Java 1.0	It is a part of the <i>java.util.concurrent</i> 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?

Internally, a HashMap uses an array of buckets to store key-value pairs. When a key-value pair is added, the key's `hashCode` is computed, and the corresponding bucket is identified using the `hashCode`. If a collision occurs (i.e., two keys have the same bucket), the HashMap resolves this by chaining (using linked lists) or using binary trees (since Java 8) if the chain length exceeds a threshold.

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.

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:

clone() – returns a copy of this object

equals() – returns true when this object is equal to the object passed as a parameter

finalize() – the garbage collector calls this method while it's cleaning the memory

getClass() – returns the runtime class of this object

hashCode() – returns a hash code of this object.

notify() – sends a notification to a single thread waiting for the object's monitor

notifyAll() – sends a notification to all threads waiting for the object's monitor

toString() – returns a string representation of this object

wait() – there are three overloaded versions of this method. It forces the current thread to wait the specified amount of time until another thread calls notify() or notifyAll() on this object.

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.

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
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 Interface:

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:

Represents 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.

```
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.

```
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

```
Callable<Void> c = () -> {
 throw new IOException("Checked exception");
};
```

---

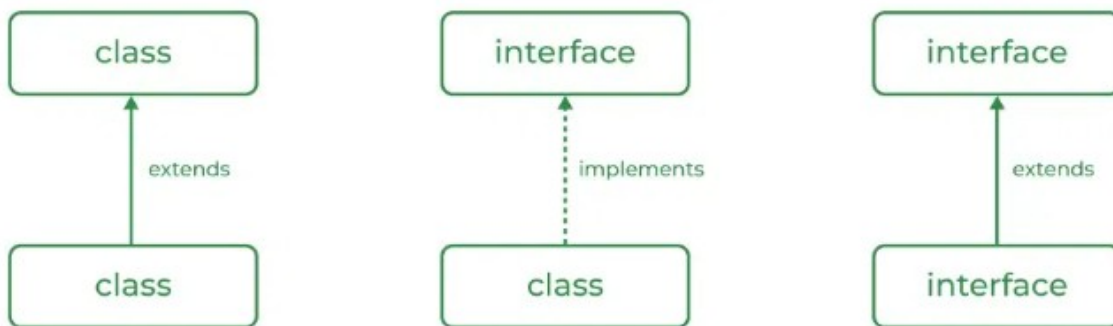
#### 47. What Are the Rules We Need to Follow When Overriding a Method That Throws an Exception?

- **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.

```
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.

```
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`.

```
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.

```
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.

```
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:**

Diamond problem occurs when two or more parent classes have methods with the same signature, leading to ambiguity about which method the subclass should inherit. Java does not allow multiple inheritance of classes however, multiple inheritance is allowed through interfaces.

To resolve it the implementing class must override the method or you can call the default method from one of the interfaces, use `InterfaceName.super.methodName()` to explicitly specify which interface's method to call.

---

**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.

```

class MyThread extends Thread {
 @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) {
 MyThread thread = new MyThread(); // Create a new thread instance
 thread.start(); // Start the thread, 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.

```

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.

```
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

```
// 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.**

---

**69.**

---

---

---

---

---

---

---

---

---

---

---

---