**Java Interview question and answer**

1. **Java 8 features:**

* Lambda Expressions: Lambda expressions enable you to treat functionality as a method argument or to create anonymous inner classes more succinctly. They facilitate functional programming by allowing you to write more concise and readable code. It does not need method name ,access modifier,return type.
* Stream API: The Stream API provides a fluent and functional approach to process collections of objects. It enables you to perform bulk operations on collections such as filtering, mapping, reducing, and sorting in a more expressive and efficient manner.
* Functional Interfaces: Java 8 introduced the @FunctionalInterface annotation to denote interfaces with a single abstract method. Functional interfaces can be used with lambda expressions, method references, and constructor references.
* **Default Methods**: Default methods allow interfaces to have methods with implementations. This feature enables the addition of methods to interfaces without breaking the classes that implement them.
* **Optional**: Optional is a container object which may or may not contain a non-null value. It helps to avoid NullPointerExceptions by encouraging the use of explicit checks for presence or absence of a value

1. **why lambda expression is used :**

**Conciseness and Readability**: Lambda expressions allow you to write more concise and readable code, especially when dealing with functional interfaces and collections. They enable you to express instances of single-method interfaces (functional interfaces) more compactly.

For example, instead of writing a full anonymous inner class implementation for a simple functional interface, you can use a lambda expression:

// Without lambda expression

Runnable runnable = new Runnable() {

@Override

public void run() {

System.out.println("Hello, World!");

}

};

// With lambda expression

Runnable runnable = () -> System.out.println("Hello, World!");

**Passing Behavior as Arguments**: Lambda expressions allow you to pass behavior (functionality) as an argument to a method. This is particularly useful when working with functional interfaces and APIs that accept behavior as parameters.

For example, in the **forEach** method of a collection, you can pass a lambda expression representing the action to be performed on each element:

List<String> names = Arrays.asList("John", "Alice", "Bob", "Jane");

names.forEach(name -> System.out.println("Hello, " + name));

Here, the lambda expression **(name -> System.out.println("Hello, " + name))** represents the behavior of printing each name with a greeting. By using lambda expressions, you can make your code more flexible and modular.

3.Singleton design pattern :

The Singleton design pattern is one of the creational design patterns and ensures that a class has only one instance and provides a global point of access to that instance. This pattern is particularly useful when exactly one object is needed to coordinate actions across the system. Here's a typical implementation of the Singleton pattern in Java:

public class Singleton {

// Private static variable to hold the single instance of the class

private static Singleton instance;

// Private constructor to prevent instantiation from outside

private Singleton() {

// Initialization code, if needed

}

// Public static method to get the single instance of the class

public static Singleton getInstance() {

// Lazy initialization: create the instance only if it's not already created

if (instance == null) {

instance = new Singleton();

}

return instance;

}

// Other methods of the class

public void doSomething() {

// Method implementation

}

}

* he class **Singleton** has a private static variable **instance** which holds the single instance of the class.
* The constructor is private, preventing instantiation of the class from outside.
* The **getInstance()** method is public and static, providing access to the single instance of the class. It uses lazy initialization, meaning the instance is created only when **getInstance()** is called for the first time.
* Other methods of the class can be accessed through the single instance returned by **getInstance()**.

Here's how you would use the Singleton class:

Singleton singleton = Singleton.getInstance();

singleton.doSomething();

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

4. Filter & aggregation in stream API

**Filtering**:

Filtering in the Stream API allows you to select elements from a stream based on certain criteria. You can use the **filter()** method to create a new stream containing only the elements that match the specified predicate.

**Aggregation**:

Aggregation operations in the Stream API allow you to compute a single result by combining elements of the stream. Common aggregation operations include summing, averaging, counting, finding the maximum or minimum, etc. You can use methods like **sum()**, **average()**, **count()**, **max()**, **min()**, etc., along with collectors to -

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

5. Stream API :

1. **Functional Approach**: Streams encourage a functional programming approach, where operations are expressed as lambda expressions or method references, making the code more concise and readable.
2. **Lazy Evaluation**: Stream operations are evaluated lazily, meaning elements are processed only when necessary. This allows for better performance and resource utilization, especially with large datasets.
3. **Parallel Processing**: Streams can leverage multi-core architectures by performing operations in parallel. Parallel streams can be created using the **parallel()** method, which splits the data into multiple chunks and processes them concurrently.
4. **Pipeline Operations**: Stream operations can be chained together to form a pipeline. Intermediate operations like **filter()**, **map()**, **sorted()**, etc., transform the stream and produce another stream, while terminal operations like **collect()**, **forEach()**, **reduce()**, etc., produce a final result or side-effect.
5. **Non-Mutating**: Stream operations do not modify the underlying data source. Instead, they produce a new stream or a final result. This immutability ensures thread-safety and facilitates easier debugging and reasoning about code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Why String is immutable?

Strings in Java are specified as immutable, as seen above because strings with the same content share storage in a single pool to minimize creating a copy of the same value. That is to say, once a String is generated, its content cannot be changed and hence changing content will lead to the creation of a new String.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

7 .Difference/similarities between Arraylist vs Linkedlist:

**ArrayList:**

1. **Underlying Data Structure**: **ArrayList** internally uses a dynamic array to store elements. When the array reaches its capacity, it is resized to accommodate more elements.
2. **Random Access**: **ArrayList** provides fast random access to elements using their indices. Retrieving an element by index takes constant time, O(1).
3. **Insertion/Deletion**: Insertion and deletion operations at the end of the list are efficient (amortized constant time, O(1)), but inserting or deleting elements in the middle of the list requires shifting subsequent elements, resulting in linear time, O(n).
4. **Memory Overhead**: **ArrayList** has a relatively low memory overhead because it only needs to store the elements and a reference to the underlying array.

**LinkedList:**

1. **Underlying Data Structure**: **LinkedList** internally uses a doubly linked list to store elements. Each element in the list contains a reference to the previous and next elements.
2. **Random Access**: **LinkedList** does not provide efficient random access to elements. Accessing an element by index requires traversing the list from the beginning or end, resulting in linear time, O(n).
3. **Insertion/Deletion**: Insertion and deletion operations at any position in the list are efficient (constant time, O(1)), as they only require updating references.
4. **Memory Overhead**: **LinkedList** has a higher memory overhead compared to **ArrayList** because each element needs to store references to the previous and next elements.

**Similarities:**

1. Both **ArrayList** and **LinkedList** implement the **List** interface, providing the same set of methods for adding, removing, and accessing elements.
2. They both allow duplicate elements and maintain the insertion order of elements.
3. Both **ArrayList** and **LinkedList** support iteration through elements using iterators or enhanced for-loops.

**Choosing Between ArrayList and LinkedList:**

* Use **ArrayList** when you need fast random access to elements or when you mostly add or remove elements at the end of the list.
* Use **LinkedList** when you frequently insert or delete elements in the middle of the list, as it provides efficient insertion and deletion operations.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

8.explain abstract and interface

**Abstract Class:**

1. **Definition**: An abstract class is a class that cannot be instantiated on its own and may contain abstract methods, concrete methods, instance variables, and constructors.
2. **Purpose**: Abstract classes are used to define common behavior and state that can be shared among multiple subclasses. They serve as a blueprint for creating concrete subclasses, which provide implementations for abstract methods.
3. **Abstract Methods**: An abstract class may contain one or more abstract methods, which are declared but not implemented in the abstract class itself. Subclasses must provide implementations for all abstract methods unless they are abstract themselves.
4. **Inheritance**: Abstract classes support inheritance, allowing subclasses to extend and specialize the behavior of the abstract class. A subclass can only extend one abstract class due to Java's single inheritance model.

**Interface:**

1. **Definition**: An interface is a reference type in Java that contains only abstract methods, default methods, static methods, constant variables (public, static, final), and nested types.
2. **Purpose**: Interfaces define a contract for classes to implement. They specify a set of methods that implementing classes must provide. Unlike abstract classes, interfaces cannot contain concrete method implementations.
3. **Abstract Methods**: All methods in an interface are implicitly abstract and public. Implementing classes must provide implementations for all interface methods.
4. **Multiple Inheritance**: Unlike abstract classes, Java supports multiple inheritance through interfaces. A class can implement multiple interfaces, allowing it to inherit behavior from multiple sources.

**Comparison**:

* Abstract classes can have constructors, instance variables, and concrete methods, while interfaces cannot.
* A class can extend only one abstract class, but it can implement multiple interfaces.
* Abstract classes are useful when you want to provide a common base implementation along with abstract methods, while interfaces are used to define contracts for classes to implement.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

9.what is try catch finally

**try**: The **try** block encloses the code that might throw an exception. It is followed by one or more **catch** blocks or a **finally** block.

**catch**: The **catch** block is used to handle exceptions that are thrown within the corresponding **try** block. It specifies the type of exception it can handle, followed by the code to execute if that exception occurs.

**finally**: The **finally** block is optional and is used to execute code that must be run regardless of whether an exception is thrown or not. This block is typically used for cleanup tasks like closing resources (e.g., file handles, database connections) that were opened in the **try** block.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

10. parallel stream

parallel stream is a feature of the Stream API introduced in Java 8 that allows for concurrent processing of elements in a stream. It enables the distribution of stream elements across multiple threads, leveraging multi-core processors to potentially speed up the processing of large data sets.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

11. inbuild methods in stream.

Intermediate operation:

Map(),filter(),sorted(),flatmap()

Terminary operation:

Foreach,collect,count.reduce

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

12. collections in java

collections represent groups of objects. They provide a way to store, retrieve, and manipulate data efficiently. Java provides a rich set of interfaces and classes for working with collections, organized in the **java.util** package. Here's an overview of some commonly used collections in Java:

1. **List**:
   * Ordered collection of elements where duplicates are allowed.
   * Common implementations: **ArrayList**, **LinkedList**, **Vector**.
   * Examples: **List<String>**, **List<Integer>**.
2. **Set**:
   * Unordered collection of unique elements.
   * Common implementations: **HashSet**, **TreeSet**, **LinkedHashSet**.
   * Examples: **Set<String>**, **Set<Integer>**.
3. **Map**:
   * Key-value pairs where keys are unique.
   * Common implementations: **HashMap**, **TreeMap**, **LinkedHashMap**.
   * Examples: **Map<String, Integer>**, **Map<Integer, String>**.
4. **Queue**:
   * Ordered collection designed for holding elements prior to processing.
   * Common implementations: **LinkedList**, **PriorityQueue**.
   * Examples: **Queue<String>**, **Queue<Integer>**.
5. **Deque**:
   * Double-ended queue that supports insertion and removal at both ends.
   * Common implementations: **ArrayDeque**, **LinkedList**.
   * Examples: **Deque<String>**, **Deque<Integer>**.
6. **Stack**:
   * Last-in, first-out (LIFO) data structure.
   * Common implementations: **Stack** (legacy class), **ArrayDeque**.
   * Examples: **Stack<String>**, **Stack<Integer>**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

13. predicate in java 8

In Java 8, the **Predicate** interface represents a functional interface that can be used to define a condition or test that evaluates to true or false. It's part of the **java.util.function** package and is commonly used in functional programming paradigms.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

14.Optional class:

In Java 8, the **Optional** class is introduced in the **java.util** package to represent an optional value, meaning it may contain a value or may be empty. This class is widely used to handle scenarios where a method may or may not return a value. It helps in avoiding **NullPointerExceptions** by explicitly indicating that a value might not be present.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

15. uses of map:

**Map** interface is used to store key-value pairs where each key is unique. It provides methods for adding, removing, and retrieving elements based on their keys. Maps are widely used in various applications for data storage, caching, indexing, and more.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

16. multithreading:

Multithreading in Java refers to the ability of a Java program to execute multiple threads concurrently. A thread is the smallest unit of execution within a process, and multithreading enables a program to perform multiple tasks simultaneously, thereby improving performance and responsiveness.

**Thread Class**: In Java, multithreading is primarily achieved using the **Thread** class. You can create a new thread by extending the **Thread** class and overriding its **run()** method, which contains the code to be executed by the thread.

**Runnable Interface**: Alternatively, you can implement the **Runnable** interface and pass an instance of your **Runnable** implementation to a **Thread** object. This approach allows you to separate the thread's behavior from its execution logic, promoting better code organization and reusability.

**Thread Pools**: Creating and managing individual threads can be resource-intensive. Thread pools, provided by the **ExecutorService** framework, allow you to manage a pool of reusable threads, improving performance and resource utilization.

**Concurrency Utilities**: Java provides a rich set of concurrency utilities in the **java.util.concurrent** package, including **ExecutorService**, **ThreadPoolExecutor**, **CountDownLatch**, **CyclicBarrier**, **Semaphore**, and **ConcurrentHashMap**, among others, to simplify multithreaded programming and address common concurrency challenges.

17.synchronisation in java

Synchronization in Java is the process of controlling access to shared resources by multiple threads to prevent data corruption, race conditions, and other concurrency-related issues. Java provides several mechanisms for synchronization, including synchronized blocks and methods,

**\**

**Synchronized Blocks and Methods**:

* The **synchronized** keyword is used to create synchronized blocks and methods in Java.
* Synchronized blocks allow you to specify a block of code that can be accessed by only one thread at a time.
* Synchronized methods are methods that are synchronized on the object's monitor (or intrinsic lock).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

18. difference between runtime and checked exceptions

**Checked Exceptions**:

* Checked exceptions are exceptions that the compiler forces you to handle explicitly.
* They are subclasses of **Exception**, excluding **RuntimeException** and its subclasses.
* Checked exceptions must be either caught using a **try-catch** block or declared in the method signature using the **throws** clause.
* Examples of checked exceptions include **IOException**, **SQLException**, and **ClassNotFoundException**.

**Runtime (Unchecked) Exceptions**:

* Runtime exceptions, also known as unchecked exceptions, are exceptions that the compiler does not enforce you to handle explicitly.
* They are subclasses of **RuntimeException** and its subclasses.
* Runtime exceptions can occur at runtime due to programming errors such as null pointer dereference, arithmetic overflow, or index out of bounds.
* While you are not required to handle them explicitly, it's still a good practice to handle or anticipate runtime exceptions, especially in critical sections of your code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

19. polymorphism

Polymorphism in Java refers to the ability of a single method or piece of code to operate on different types of objects. It allows objects of different classes to be treated as objects of a common superclass, providing flexibility and extensibility in object-oriented programming. There are two main types of polymorphism in Java: compile-time (static) polymorphism and runtime (dynamic) polymorphism.

1. **Compile-Time Polymorphism**:
   * Also known as static polymorphism or method overloading.
   * Occurs when the compiler determines which method to invoke based on the method signature at compile time.
   * Methods with the same name but different parameters or return types can be overloaded within the same class.

**Runtime Polymorphism**:

* Also known as dynamic polymorphism or method overriding.
* Occurs when the method to be invoked is determined at runtime based on the actual type of the object.
* Involves subclassing and overriding methods defined in the superclass.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

20. What is method reference.

Method reference in Java is a shorthand syntax for referring to methods or constructors using the **::** operator. It allows you to treat a method as a lambda expression or pass it as an argument to a method that expects a functional interface. Method references make code more concise and readable, especially when working with functional interfaces.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

21. try with resources, how many class you can create using with try with resources

In Java, the **Try-with-resources**statement is a try statement that declares one or more resources in it. A resource is an object that must be closed once your program is done using it. For example, a File resource or a Socket connection resource.  The try-with-resources statement ensures that each resource is closed at the end of the statement execution. If we don’t close the resources, it may constitute a resource leak and also the program could exhaust the resources available to it.

You can pass any object as a resource that implements *java.lang.AutoCloseable*, which includes all objects which implement java.io.Closeable.

By this, now we don’t need to add an extra [finally block](https://www.geeksforgeeks.org/g-fact-24-finalfinally-and-finalize-in-java/) for just passing the closing statements of the resources. The resources will be closed as soon as the try-catch block is executed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

22. searlization, deserialization

Serialization and deserialization are processes in Java used for converting an object into a stream of bytes to persist or transmit it over a network, and then reconstructing the object from the serialized bytes.

**Serialization**: Serialization is the process of converting an object into a byte stream so that it can be stored into a file, sent over a network, or saved in a database. In Java, serialization is achieved by implementing the **Serializable** interface, which is a marker interface that indicates that a class can be serialized. When an object of a serializable class is serialized, all of its non-transient fields are converted into a byte stream.

**Deserialization**: Deserialization is the process of converting a byte stream back into an object. In Java, deserialization is achieved by reading the byte stream and reconstructing the object from it. The class of the object being deserialized must be available in the classpath. During deserialization, the JVM creates a new object and initializes its fields from the byte stream.

23. types of string declaration and how it stores internally

there are two types of string declaration:

1. **Using String Literal**:

When a string is declared using string literals (enclosed in double quotes), Java creates a string object in the string pool if it doesn't already exist. If another string with the same value already exists in the pool, the newly created string references the existing one. String literals are stored in the string pool, a special area of the heap memory.

**Using the new Keyword**:

When a string is created using the **new** keyword, a new string object is created in the heap memory, regardless of whether the same string value already exists in the string pool. This method is less memory-efficient compared to using string literals.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

24. string intern

In Java, the **intern()** method is used to add a string to the string pool and return a reference to the interned string. When you call the **intern()** method on a string object, Java checks if the string already exists in the string pool. If the string is not present in the pool, it adds the string to the pool and returns a reference to the interned string. If the string is already present in the pool, it returns a reference to the existing interned string.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

25. heap vs constat pool

1. Heap memory
   * The heap is a region of memory used for dynamic memory allocation. It is where objects and arrays are allocated when your Java program runs.
   * Objects created with the **new** keyword, as well as arrays, are stored in the heap.
   * Each Java application has one heap, which is shared among all threads running in the application.
   * The heap is managed by the Java Virtual Machine (JVM), which is responsible for allocating and deallocating memory as needed.
   * Objects stored in the heap are subject to garbage collection, meaning that memory occupied by objects no longer in use can be rec
   * laimed by the JVM.
2. **Constant Pool**:
   * The constant pool is a special area of memory within the heap that is used to store string literals, class and interface names, method and field names, and other constants.
   * When you declare a string literal in your Java code (e.g., **"hello"**), it is automatically added to the constant pool.
   * Additionally, class and interface names, method and field names, and other constants used in your code are stored in the constant pool.
   * The constant pool is used by the JVM to optimize memory usage and improve performance by reusing existing constants rather than creating new ones each time they are needed.
   * The constant pool is part of the class file format and is loaded into memory when a class is loaded by the JVM.
   * Unlike the heap, the constant pool is not subject to garbage collection because its contents are fixed at compile time and cannot be modified at runtime.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

26. what is getter & setter method?

Getter and setter methods are a pair of functions commonly used in object-oriented programming to access and modify the attributes or properties of an object, respectively.

**Getter Method**: A getter method, also known as an accessor method, is used to retrieve the value of a private or protected attribute of an object. It typically has the following characteristics:

* It is a public method.
* It does not modify the state of the object.
* It returns the value of a private variable.

**Setter Method**: A setter method, also known as a mutator method, is used to modify the value of a private or protected attribute of an object. It typically has the following characteristics:

* It is a public method.
* It updates the state of the object by modifying the value of a private variable.
* It does not return any value (void).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

27. how we make class as immutable?

1. **Make the class final**: This prevents the class from being extended, ensuring that its behavior cannot be modified by subclassing.
2. **Make fields private and final**: This ensures that the state of the object cannot be changed after construction.
3. **Remove setter methods**: Eliminate methods that modify the state of the object after construction, including setter methods.
4. **Ensure deep immutability for mutable fields**: If the class contains mutable fields (e.g., collections), ensure that they are defensively copied or made immutable themselves to prevent external modification.
5. public List<String> getImmutableList() {
6. return Collections.unmodifiableList(immutableList);

}

28. Can you brief polymorphism and encapsulation

1. **Polymorphism**:

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class or interface. It enables the use of a single interface to represent different types of objects. Polymorphism can be achieved through two main mechanisms:

* + **Compile-time Polymorphism**: Also known as static polymorphism, this occurs when the method to be invoked is determined at compile time. Examples include method overloading, where multiple methods with the same name but different parameters exist in the same class, and operator overloading.
  + **Runtime Polymorphism**: Also known as dynamic polymorphism, this occurs when the method to be invoked is determined at runtime. It is achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

Polymorphism enhances code reusability, flexibility, and maintainability by allowing methods to be invoked based on the actual type of object at runtime.

1. **Encapsulation**:

Encapsulation is a fundamental principle of OOP that involves bundling data (attributes) and methods (behaviors) that operate on the data into a single unit called a class. It hides the internal state of an object and restricts access to it from outside the class, providing controlled access through methods.

Encapsulation achieves several objectives:

* + **Data Hiding**: It prevents direct access to an object's internal state, protecting it from unauthorized modification and ensuring data integrity.
  + **Abstraction**: It exposes only essential information about an object's behavior while hiding implementation details. This simplifies the usage of the class and promotes modularity.
  + **Access Control**: It allows defining access modifiers (e.g., public, private, protected) to control the visibility of attributes and methods, enabling fine-grained access control and ensuring encapsulation.

Encapsulation enhances code maintainability, reusability, and reliability by encapsulating implementation details, reducing coupling between classes, and promoting a clear separation of concerns.

Top of Form

29. What is interface why we use that

An interface in programming is a reference type that defines a set of abstract methods that a class must implement. It establishes a contract for what a class can do, without specifying how it does it. Here's why interfaces are used:

1. **Abstraction**: Interfaces provide a way to achieve abstraction in programming by defining a set of methods that represent a behavior or capability. They hide the implementation details and only expose the functionality that is relevant to the interaction.
2. **Multiple Inheritance**: Unlike classes, which support single inheritance in most object-oriented languages, interfaces allow for multiple inheritance. A class can implement multiple interfaces, inheriting behaviors from each one.
3. **Polymorphism**: Interfaces enable polymorphism, allowing objects of different classes to be treated interchangeably if they implement the same interface. This facilitates code reuse and flexibility.
4. **Contract**: Interfaces define a contract between the interface and the implementing class. Any class that implements an interface must provide concrete implementations for all the methods declared in the interface, thus fulfilling the contract.
5. **Decoupling**: Interfaces promote loose coupling between components of a system. Classes that interact with each other through interfaces are less dependent on each other's concrete implementations, making the system more modular and easier to maintain.
6. **API Design**: Interfaces are commonly used in API design to define the public contract of a library or framework. They provide a clear specification of the available functionality and how it can be accessed by client code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

30. Fail fast and fail safe

**Fail-Fast Iterators**:

In Java, many collections, such as **ArrayList**, **HashMap**, and **HashSet**, use fail-fast iterators by default. Fail-fast iterators immediately throw a **ConcurrentModificationException** if the underlying collection is structurally modified (i.e., elements are added or removed) while an iterator is traversing it. This behavior ensures that the iterator detects any concurrent modifications and prevents the possibility of unpredictable behavior or data corruption.

**Fail-Safe Iterators**:

While fail-fast iterators provide immediate detection of concurrent modifications, fail-safe iterators, on the other hand, do not throw exceptions if the underlying collection is modified during iteration. Instead, they operate on a copy of the collection's data (typically created when the iterator is instantiated). This ensures that the original collection remains unchanged, and the iteration proceeds without interference from modifications made by other threads.

31. Explain Hashmap and hash set

**HashMap**:

* **HashMap** is a data structure that stores elements in key-value pairs.
* It uses a hash table internally to store the elements, which allows for fast retrieval, insertion, and deletion operations (assuming a good hash function).
* Each key in a **HashMap** must be unique. However, **HashMap** allows **null** as a key and value.
* It does not maintain the order of elements. The order in which elements are stored is not guaranteed and can change over time.
* **HashMap** provides constant-time performance for basic operations (**get**, **put**, **remove**) on average, assuming the hash function disperses the elements properly among the buckets.

**HashSet**:

* **HashSet** is an implementation of the **Set** interface that uses a hash table for storage.
* It does not allow duplicate elements. Adding duplicate elements to a **HashSet** will have no effect.
* Like **HashMap**, **HashSet** does not maintain the order of elements. The order is not guaranteed and can change over time.
* It provides constant-time performance for basic operations (**add**, **contains**, **remove**) on average, assuming a good hash function.
* **HashSet** allows **null** as an element.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

32. What is purpose of default method in interface

**Backward Compatibility**: Default methods enable adding new methods to interfaces without breaking existing implementations. Classes that implement the interface are not required to provide an implementation for the default method unless they choose to override it.

**Code Reuse**: Default methods allow for code reuse across multiple classes that implement the same interface. If multiple classes share common behavior, it can be encapsulated in a default method in the interface, reducing code duplication.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

33. Ways to achieve polymorphism

\* Method overloading

\*Method overriding

\*Interface implementation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

34. List vs Map

1. **List**:
   * A List is an ordered collection of elements that allows duplicate elements.
   * Elements in a List are stored based on their insertion order.
   * Lists are accessed and indexed by position, starting from index 0.
   * Main implementations: ArrayList, LinkedList, Vector.
   * Example use cases: Maintaining a sequence of elements, implementing a stack or a queue, representing a collection of items to display in a specific order.
2. **Map**:
   * A Map is a collection that maps keys to values, where each key is unique and corresponds to a single value.
   * Maps do not allow duplicate keys; each key-value pair is unique.
   * Maps are not ordered; the order of elements is not guaranteed and may vary between iterations.
   * Main implementations: HashMap, TreeMap, LinkedHashMap.
   * Example use cases: Associating data with unique identifiers (keys), implementing lookup tables, caching key-value pairs, storing configuration settings.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

35. Hash table, Hashing internal working

A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets, from which the desired value can be found. The hash function transforms the key into a hash code, which is then used to determine the index in the array where the value associated with the key should be stored or retrieved.

Here's a brief overview of how a hash table works internally:

1. **Hash Function**:
   * A hash function takes a key as input and produces a hash code, which is typically an integer value.
   * The hash code is computed deterministically based on the contents of the key. Ideally, the hash function evenly distributes keys across the available array indices to minimize collisions.
2. **Array of Buckets**:
   * A hash table internally consists of an array of buckets, also known as slots or bins.
   * Each bucket can hold one or more key-value pairs, depending on the implementation.
3. **Hash Code to Index Mapping**:
   * The hash code produced by the hash function is used to determine the index of the bucket where the key-value pair should be stored or retrieved.
   * The hash code is typically reduced to fit within the range of array indices using a modulo operation with the size of the array.
4. **Handling Collisions**:
   * Collisions occur when multiple keys map to the same index in the array.
   * Hash tables use collision resolution techniques to handle collisions and store multiple key-value pairs in the same bucket. Common techniques include:
     + Chaining: Each bucket contains a linked list (or another data structure) of key-value pairs that hash to the same index.
     + Open Addressing: If a collision occurs, the algorithm probes for an empty slot (by a specific probing sequence) in the array to store the collided key-value pair.
5. **Retrieving Values**:
   * To retrieve a value associated with a key, the hash function is applied to the key to compute the hash code.
   * The hash code is used to determine the index of the bucket where the value is stored.
   * If collisions occur, the appropriate collision resolution technique is used to locate the desired value.
6. **Dynamic Resizing**:
   * Hash tables often dynamically resize themselves to maintain a reasonable load factor (the ratio of the number of elements to the number of buckets).
   * When the load factor exceeds a certain threshold, the array of buckets is resized (usually doubled in size) and all key-value pairs are rehashed and redistributed into the new array.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

36.Ways to break singleton design pattern?

1. **Multiple Class Loaders**: If a class is loaded by multiple class loaders in a Java application, it can result in multiple instances of the Singleton class. Each class loader maintains its own namespace, so the Singleton class loaded by one class loader will be different from the one loaded by another class loader.
2. **Reflection**: Reflection in Java allows accessing and modifying fields, methods, and constructors of classes at runtime. Using reflection, it's possible to access the private constructor of the Singleton class and create new instances, thereby breaking the Singleton pattern.
3. **Serialization and Deserialization**: If a Singleton class implements the **Serializable** interface and is serialized and then deserialized, it can result in multiple instances of the class. During deserialization, the object is reconstructed, bypassing the constructor, which can create a new instance.
4. **Cloning**: If a Singleton class overrides the **clone()** method of the **Object** class and allows cloning, it's possible to create a new instance of the Singleton class by cloning the existing instance. To prevent this, you can throw an exception in the **clone()** method or override it to return the existing instance.
5. **Multithreading Issues**: In a multithreaded environment, if the Singleton class is not properly synchronized and multiple threads simultaneously access the **getInstance()** method, it can lead to the creation of multiple instances. Double-checked locking is a common pattern used to address this issue.

**Garbage Collection**: If the Singleton instance can be garbage collected, subsequent calls to **getInstance()** may return a new instance. To prevent this, you can use a strong reference to hold the Singleton instance or use a different instantiation approach like eager initialization.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

37.purpose of singleton design pattern

1. **Resource Sharing**: Singleton pattern is used when a single resource, such as a database connection, file system, or logger, needs to be shared among multiple parts of the system. By ensuring that only one instance of the resource exists, the pattern helps in efficient resource utilization and management.
2. **Global Configuration Settings**: Singleton pattern can be used to manage global configuration settings that need to be accessed and modified by different parts of the application. For example, application settings related to logging levels, caching mechanisms, or user preferences can be encapsulated in a Singleton object.
3. **Caching**: Singletons can be used to implement caching mechanisms where a single cache instance is shared across different components of the system. This helps in improving performance by reducing redundant computations and database accesses.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

38. is it possible to override the static method?

No, it's not possible to override a static method in Java. Static methods are associated with the class itself rather than with any particular instance of the class. They belong to the class's definition rather than to any instance of the class.

When a subclass defines a static method with the same signature as a static method in its superclass, it is not considered overriding; instead, it is called "method hiding". In other words, the subclass provides a new definition for the static method that is independent of the superclass's static method.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

39. Rest api status code.

1. **1xx - Informational**:
   * **100 Continue**: The server has received the request headers and the client should proceed with the request.
2. **2xx - Success**:
   * **200 OK**: The request has succeeded.
   * **201 Created**: The request has been fulfilled, and a new resource has been created.
   * **204 No Content**: The server successfully processed the request, but there is no content to return.
3. **3xx - Redirection**:
   * **301 Moved Permanently**: The requested resource has been permanently moved to a new location.
   * **304 Not Modified**: The resource has not been modified since the last request, and the client should use the cached data.
4. **4xx - Client Error**:
   * **400 Bad Request**: The server cannot process the request due to a client error, such as malformed syntax or invalid parameters.
   * **401 Unauthorized**: The request requires user authentication.
   * **403 Forbidden**: The server understood the request but refuses to authorize it.
   * **404 Not Found**: The requested resource could not be found.
5. **5xx - Server Error**:
   * **500 Internal Server Error**: A generic error message indicating that the server encountered an unexpected condition that prevented it from fulfilling the request.
   * **502 Bad Gateway**: The server received an invalid response from an upstream server while trying to fulfill the request.
   * **503 Service Unavailable**: The server is currently unable to handle the request due to temporary overloading or maintenance.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

40. ConcurrentModification Exception

A **ConcurrentModificationException** in Java typically occurs when you try to modify a collection (such as a **List**, **Set**, or **Map**) while iterating over it using an iterator, and another thread modifies the collection structurally (by adding or removing elements) at the same time.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

41. Throw and Throws

**throw**:

* **throw** is a keyword used to explicitly throw an exception within a method or block of code.
* It is followed by an instance of an exception class or a subclass of **Throwable**.
* When the **throw** statement is executed, the control immediately jumps to the nearest enclosing **try** block's **catch** clauses to handle the thrown exception.

**throws**:

* **throws** is used in the method signature to indicate that the method may throw one or more types of exceptions during its execution.
* It is followed by a comma-separated list of exception classes.
* It does not handle the exceptions; it simply declares that the method might throw those exceptions, and it is the responsibility of the caller to handle them or propagate them further.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

42. Default vs Static Methods in Java8

**Default Methods**:

* Default methods are methods in an interface that have a default implementation.
* They are declared using the **default** keyword.
* Default methods allow interfaces to provide method implementations without requiring the implementing classes to provide an implementation.
* They enable adding new methods to interfaces in a backward-compatible manner.
* Default methods can be overridden by classes that implement the interface, but it's not mandatory to override them.
* They are mainly used for providing backward compatibility in interfaces, especially when new methods are added to an existing interface.

**Static Methods**:

* Static methods in interfaces are similar to static methods in classes.
* They are declared using the **static** keyword.
* Static methods belong to the interface and can be called using the interface name.
* They cannot be overridden in implementing classes, as they are not instance methods.
* Static methods provide utility methods that are common to all implementations of the interface.
* They cannot access instance variables directly because they do not belong to any instance.

43. Use Streams to print the even numbers

List<Integer>num=Arrays.*asList*(1,2,3,4,5,6,5,8,5,786,89,565,6,8,56);

num.stream().filter(x -> x%2==0).distinct().forEach(System.***out***::println);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

44. Int vs Integer

1. **Primitive Type vs Wrapper Class**:
   * **int** is a primitive data type.
   * **Integer** is a wrapper class for the primitive type **int**.
2. **Nullability**:
   * **int** is a primitive type and cannot be **null**.
   * **Integer** is an object and can be **null**.
3. **Performance**:
   * Accessing and operating on **int** values is generally faster than **Integer** due to the overhead of object creation and autoboxing/unboxing associated with **Integer**.
4. **Usage**:
   * **int** is typically used when you need a simple integer value and do not require nullability or additional methods provided by **Integer**.
   * **Integer** is used when you need to represent an integer value as an object, for example, when working with collections or APIs that require objects rather than primitives.
5. **Collections**:
   * Java collections (e.g., **List**, **Set**, **Map**) can only store objects, not primitives. Therefore, if you need to store integer values in collections, you must use **Integer** instead of **int**.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

45. In which scenario will go for abstract and when to interface?

**Abstract Class:**

1. **When you want to provide a default implementation**: Abstract classes can contain both abstract and concrete methods. If you have methods with default behavior that should be inherited by subclasses, you can define them in an abstract class.
2. **When you want to share code among closely related classes**: Abstract classes are useful when you have a group of closely related classes that share some common functionality. You can implement common methods in the abstract class to avoid code duplication in subclasses.
3. **When you want to define instance variables**: Abstract classes can have instance variables, constructors, and other features that interfaces cannot have.
4. **When you want to evolve your API without breaking existing implementations**: Adding new methods to an abstract class is backward compatible because existing subclasses are not required to implement them.

**Interface:**

1. **When you want to define a contract for unrelated classes**: Interfaces are ideal for defining contracts that unrelated classes can implement. If you want to specify a set of methods that multiple classes should implement, interfaces provide a way to do so.
2. **When you want to achieve multiple inheritances of type**: Unlike classes, Java doesn't support multiple inheritance of implementation. However, a class can implement multiple interfaces. This makes interfaces a good choice when you want to achieve polymorphism with unrelated classes.
3. **When you want to define behavior for a group of related classes**: Interfaces can be used to define a common behavior for a group of related classes without providing a default implementation. This allows for more flexibility and avoids tight coupling between classes.
4. **When you want to define a lightweight contract**: Interfaces provide a lightweight way to define contracts without introducing the complexity of inheritance that comes with abstract classes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

46. Comparator vs Comparable

1. **Comparable**:
   * The **Comparable** interface is used to define the natural ordering of objects.
   * Objects that implement the **Comparable** interface can be compared to other objects of the same type using the **compareTo()** method.
   * The **compareTo()** method returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object, respectively.
   * When a class implements **Comparable**, it specifies how instances of that class should be compared to each other.
2. **Comparator**:
   * The **Comparator** interface is used to define custom comparison logic separate from the natural ordering of objects.
   * It provides a way to sort objects based on different criteria without modifying their original implementation.
   * Objects that implement the **Comparator** interface can be used to compare two objects of a specific type using the **compare()** method.
   * The **compare()** method returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second object, respectively.
   * **Comparator** instances can be used to sort elements of a collection, and multiple comparators can be defined to sort objects based on different criteria.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

47. Difference between filter and map in stream?

1. **filter**:
   * The **filter** operation is used to select elements from a stream based on a specified criteria.
   * It takes a **Predicate** as an argument, which is a functional interface representing a boolean-valued function of one argument.
   * The **filter** operation evaluates the predicate for each element of the stream and retains only those elements for which the predicate returns **true**.
   * It returns a new stream containing only the elements that satisfy the given predicate.
2. **map**:
   * The **map** operation is used to transform each element of a stream into another object.
   * It takes a **Function** as an argument, which is a functional interface representing a function that accepts one argument and produces a result.
   * The **map** operation applies the specified function to each element of the stream and produces a new stream consisting of the results of applying the function to each element.
   * It returns a new stream containing the transformed elements.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

48. write a program using streams to sort the employee based on there joining date.

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.List;

class Employee {

private String name;

private LocalDate joiningDate;

public Employee(String name, LocalDate joiningDate) {

this.name = name;

this.joiningDate = joiningDate;

}

public String getName() {

return name;

}

public LocalDate getJoiningDate() {

return joiningDate;

}

@Override

public String toString() {

return "Employee{name='" + name + "', joiningDate=" + joiningDate + '}';

}

}

public class Main {

public static void main(String[] args) {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee("Alice", LocalDate.of(2018, 5, 15)));

employees.add(new Employee("Bob", LocalDate.of(2017, 10, 8)));

employees.add(new Employee("Charlie", LocalDate.of(2019, 3, 21)));

// Sort employees based on their joining dates using streams

List<Employee> sortedEmployees = employees.stream()

.sorted(Comparator.comparing(Employee::getJoiningDate))

.toList();

// Display sorted employees

System.out.println("Employees sorted by joining date:");

sortedEmployees.forEach(System.out::println);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

49. In a final List can we add or remove the values? Why

In Java, the **final** keyword when applied to a variable indicates that the variable's reference cannot be changed once it's initialized. However, it does not mean that the object itself is immutable.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

50.Tranciant keyword

transient is a variables modifier used in serialization. At the time of serialization, if we don't want to save value of a particular variable in a file, then we use transient keyword. When JVM comes across transient keyword, it ignores original value of the variable and save default value of that variable data type

51.Volatile key word

The volatile keyword in Java is used to indicate that a variable's value can be modified by different threads.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

52. Garbage Collection internal working

Garbage collection (GC) is a process in Java that automatically manages memory by reclaiming memory that is no longer in use by the program. The internal working of garbage collection involves several steps and algorithms to identify and reclaim unreachable objects. Here's an overview of how garbage collection works internally in Java:

1. **Mark and Sweep**:
   * The most common garbage collection algorithm used in Java is the mark and sweep algorithm.
   * It consists of two phases: marking and sweeping.
   * During the marking phase, the garbage collector traverses the object graph starting from the root objects (such as global variables, local variables, and static variables) and marks all reachable objects as live.
   * In the sweeping phase, the garbage collector scans the entire heap and reclaims memory occupied by objects that are not marked as live (i.e., unreachable objects).
2. **Root Set**:
   * The root set consists of all objects that are directly accessible by the executing threads, such as local variables, static variables, and references from active threads.
   * The garbage collector starts traversing the object graph from the root set to identify all reachable objects.
3. **Tracing**:
   * During the marking phase, the garbage collector uses a process called tracing to follow references from live objects to other objects in memory.
   * It traverses object references recursively, marking each visited object as live until all reachable objects are marked.
4. **Reference Counting**:
   * Another approach to garbage collection is reference counting, where each object keeps track of the number of references to it.
   * When an object's reference count drops to zero, it is considered unreachable and eligible for garbage collection.
   * However, Java does not use reference counting for garbage collection because it cannot handle circular references efficiently and has performance overhead for every reference update.
5. **Generational Garbage Collection**:
   * In addition to mark and sweep, modern JVM implementations, such as the HotSpot JVM, use generational garbage collection.
   * Generational garbage collection divides the heap into different generations (young generation, old generation, and permanent generation).
   * It leverages the observation that most objects become unreachable shortly after they are allocated.
   * Young generation garbage collection (minor GC) is more frequent and efficient, while old generation garbage collection (major GC) is less frequent and more time-consuming.
6. **Garbage Collector Types**:
   * Java has different garbage collector implementations, such as Serial GC, Parallel GC, CMS (Concurrent Mark-Sweep) GC, G1 (Garbage-First) GC, and ZGC (Z Garbage Collector).
   * Each garbage collector has its own algorithms, trade-offs, and configurations for optimizing different application workloads and performance characteristics.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

53. Object comparison Types of Objects comparison

In Java, object comparison refers to the process of determining whether two objects are considered equal or not based on certain criteria. There are several ways to compare objects in Java, depending on the requirements and the nature of the objects being compared. Here are some common approaches:

1. **Using equals() Method**:
   * The **equals()** method is defined in the **Object** class and can be overridden by classes to provide custom equality comparison logic.
   * By default, the **equals()** method in the **Object** class performs reference equality comparison (i.e., checks if two object references point to the same memory location).
   * It's a common practice to override the **equals()** method in custom classes to provide value-based equality comparison. This typically involves comparing the internal state of objects.
   * When overriding **equals()**, it's important to adhere to the general contract of the method, which specifies properties such as reflexivity, symmetry, transitivity, and consistency.
2. **Using == Operator**:
   * The **==** operator in Java is used for reference equality comparison, meaning it checks whether two object references point to the same memory location.
   * When using the **==** operator to compare objects, you're comparing the memory addresses of the objects, not their contents.
3. **Using compareTo() Method**:
   * For objects that implement the **Comparable** interface, such as **String**, **Integer**, and other wrapper classes, you can use the **compareTo()** method to compare objects for ordering.
   * The **compareTo()** method returns a negative integer, zero, or a positive integer depending on whether the current object is less than, equal to, or greater than the specified object.
4. **Custom Comparison Logic**:
   * For complex objects or cases where the default comparison methods (**equals()**, **==**, **compareTo()**) are not suitable, you can define custom comparison logic by implementing a comparator or using custom comparison methods.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

54. What is deadlock situation and how to avoid it

A deadlock situation occurs in a multithreaded environment when two or more threads are blocked indefinitely, waiting for each other to release resources that they need. Deadlocks can occur when multiple threads acquire locks on resources in different orders, leading to a cyclic dependency where each thread is waiting for a resource held by another thread.

To illustrate deadlock, consider the following scenario involving two threads (**Thread A** and **Thread B**) and two resources (**Resource 1** and **Resource 2**):

1. **Thread A** acquires **Resource 1**.
2. **Thread B** acquires **Resource 2**.
3. **Thread A** tries to acquire **Resource 2** but gets blocked because it's held by **Thread B**.
4. **Thread B** tries to acquire **Resource 1** but gets blocked because it's held by **Thread A**.

Now, both threads are blocked, waiting for resources held by each other, resulting in a deadlock situation.

To avoid deadlocks, you can follow several strategies:

1. **Lock Ordering**:
   * Ensure that all threads acquire locks in a consistent and predefined order to prevent cyclic dependencies.
   * Assign a total ordering to locks and always acquire locks in increasing order or decreasing order.
2. **Lock Timeout**:
   * Use lock timeouts or try-lock mechanisms to acquire locks with a timeout period.
   * If a thread cannot acquire a lock within the specified timeout, it can release the held locks and retry later, avoiding potential deadlocks.
3. **Avoid Nested Locks**:
   * Minimize the use of nested locks or nested synchronized blocks to reduce the likelihood of deadlocks.
   * If a thread already holds a lock, try to avoid acquiring additional locks while holding it.
4. **Lock Hierarchy**:
   * Establish a clear lock hierarchy and always acquire locks in a hierarchical order.
   * Threads should acquire locks only if they are at the highest level of the lock hierarchy or are at a compatible level.
5. **Resource Allocation Order**:
   * Allocate resources in a fixed and predefined order across all threads to avoid potential conflicts.
   * Ensure that threads always request resources in the same order to prevent cyclic dependencies.
6. **Deadlock Detection and Recovery**:
   * Implement deadlock detection mechanisms to periodically check for deadlock conditions in the system.
   * Upon detecting a deadlock, take appropriate actions such as releasing locks, aborting transactions, or restarting threads to recover from the deadlock situation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

55. can we use static with transient keyword?

No, it's not possible to use the **static** keyword with the **transient** keyword in Java. These two keywords serve different purposes and are mutually exclusive in their usage.

Here's what each keyword does:

1. **static**: When applied to a variable, method, or nested class, the **static** keyword indicates that the member belongs to the class itself rather than to instances of the class. It means there is only one copy of the static member shared among all instances of the class. Static members can be accessed directly using the class name without the need to create an instance of the class.
2. **transient**: When applied to an instance variable, the **transient** keyword instructs the Java serialization mechanism to exclude the variable from being serialized. In other words, transient variables are not saved as part of the object's state when the object is serialized. When the object is deserialized, transient variables are initialized with their default values.

56. If a request is hitting in your DB frequently which collection will be used.

If a request is hitting your database frequently, and you need to store frequently accessed data in memory to improve performance, you might consider using a caching mechanism rather than a collection directly. However, if you need to choose a collection for storing data in memory, the choice depends on several factors such as the nature of the data, access patterns, concurrency requirements, and memory constraints. Here are some considerations:

1. **HashMap**:
   * Use a **HashMap** if you need fast lookup by key. HashMap provides constant-time performance for the **get(key)** and **put(key, value)** operations on average.
   * HashMap is suitable for cases where you have key-value pairs and need efficient retrieval by key.
2. **ConcurrentHashMap**:
   * If your application has multiple threads accessing the collection concurrently, consider using **ConcurrentHashMap**. It provides thread-safe access without requiring explicit synchronization.
   * ConcurrentHashMap is suitable for concurrent read and write access scenarios where thread safety is essential.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**57.** **What is difference between normal forEach and streams forEach.**

**Normal forEach**:

* The traditional **forEach** loop is used to iterate over elements in a collection or array.
* It is implemented using an enhanced for loop or a traditional for loop.
* The syntax is straightforward and familiar to Java developers.
* It iterates over elements sequentially and performs the specified action for each element.
* Normal **forEach** is not parallelizable and cannot take advantage of multi-core processors for concurrent processing.

**Streams forEach**:

* The **forEach** method provided by streams is used to iterate over elements in a stream.
* It is part of the Stream API introduced in Java 8.
* It operates on streams and allows for functional-style iteration and processing of elements.
* It can be used in sequential streams as well as parallel streams, allowing for concurrent processing of elements.
* The **forEach** method in streams is often used as part of a stream pipeline to perform transformations, filtering, and other operations on elements.

58.Heap memory vs stack memory

1. **Stack Memory**:
   * Stack memory is used for storing method call frames and local variables.
   * Each thread in a Java application has its own stack memory.
   * Stack memory is organized in a Last-In-First-Out (LIFO) manner, meaning that the most recently called method is at the top of the stack, and the method that called it is below it, and so on.
   * Stack memory is fast and efficient because it is allocated and deallocated automatically as method calls are made and returned.
   * Stack memory size is typically fixed and limited, and it is smaller compared to heap memory.
   * Stack memory is thread-safe because each thread has its own stack.
2. **Heap Memory**:
   * Heap memory is used for dynamic memory allocation and object storage.
   * It is a shared resource accessible to all threads in a Java application.
   * Objects created in Java are stored in the heap memory.
   * Heap memory is managed by the Java Virtual Machine (JVM) through garbage collection, which automatically deallocates memory for objects that are no longer in use.
   * Heap memory size can grow dynamically based on the demand for memory by the application.
   * Heap memory is not thread-safe, and care must be taken to synchronize access to shared objects in multi-threaded environments.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

59. Whether the class can be static?

Java, classes themselves cannot be declared as **static**. However, you can declare members of a class as **static**, including fields, methods, and nested classes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

60.String buffer and String builder

1. **StringBuffer**:
   * **StringBuffer** was introduced in Java 1.0.
   * It is thread-safe, meaning that it is synchronized, and its methods are inherently synchronized. This ensures that multiple threads can safely manipulate a **StringBuffer** object without causing data corruption or inconsistency.
   * The synchronization overhead in **StringBuffer** can lead to a performance penalty, especially in multi-threaded environments where synchronization is not always necessary.

**StringBuilder**:

* + **StringBuilder** was introduced in Java 5.0 as part of the Java Collections Framework.
  + It is not thread-safe, meaning that its methods are not synchronized. This makes **StringBuilder** more efficient in single-threaded environments or situations where thread safety is not a concern.
  + Because it is not synchronized, **StringBuilder** does not incur the synchronization overhead associated with **StringBuffer**, resulting in potentially better performance, especially in scenarios where concurrent access is not required.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

61. Are the methods in object class are final?

No, the methods in the **Object** class are not all **final**. The **Object** class is the root class for all classes in Java, and it provides several methods that are commonly overridden by subclasses. Some of these methods are not **final**, meaning that subclasses can override them to provide custom implementations as needed.

Here are some commonly overridden methods in the **Object** class that are not **final**:

1. **equals(Object obj)**: Compares this object with another object for equality.
2. **hashCode()**: Returns a hash code value for the object.
3. **toString()**: Returns a string representation of the object.
4. **clone()**: Creates and returns a copy of this object.
5. **finalize()**: Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

However, there are also some methods in the **Object** class that are **final**, meaning that they cannot be overridden by subclasses. These include:

1. **getClass()**: Returns the runtime class of this object.
2. **wait(), wait(long timeout), wait(long timeout, int nanos), notify(), and notifyAll()**: Methods used for inter-thread communication and synchronization.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

62. How will you read a file from SFTP ?

To read a file from an SFTP (SSH File Transfer Protocol) server in Java, you can use the JSch library, which is a pure Java implementation of SSH2. JSch allows you to connect to SSH servers, including SFTP servers, and perform file operations like reading, writing, and transferring files securely over the SSH protocol.