Core Java

1. Collections. Diff between ArrayList and Set (real world scenario where you have used)

| **Feature** | **ArrayList** | **Set** |
| --- | --- | --- |
| **Duplicates** | Allows duplicates | Does **not** allow duplicates |
| **Order** | Maintains insertion order | Order depends on the implementation (HashSet - Unordered, TreeSet - Sorted, LinkedHashSet - Insertion order) |
| **Index-Based Access** | Yes, can access elements using get(index) | No, elements are accessed via iteration |
| **Performance** | Faster for indexed access | Generally faster for unique element checks |
| **Implementation Classes** | ArrayList<E> | HashSet<E>, LinkedHashSet<E>, TreeSet<E> |
| **Best Use Case** | When duplicate elements and ordered access are needed | When only unique elements are needed |

**Scenario: Storing Customer Emails**

* **ArrayList Usage**: If you need to maintain a list of customer emails **in the order** they signed up (even if there are duplicates).
* **Set Usage**: If you want to store only **unique** customer emails (avoiding duplicate registrations).

**Example with ArrayList (Duplicates Allowed)**

ArrayList<String> emails = new ArrayList<>();

emails.add("alice@example.com");

emails.add("bob@example.com");

emails.add("alice@example.com"); // Duplicate allowed

System.out.println(emails); // Output: [alice@example.com, bob@example.com, alice@example.com]

**Example with Set (No Duplicates)**

HashSet<String> emails = new HashSet<>();

emails.add("alice@example.com");

emails.add("bob@example.com");

emails.add("alice@example.com"); // Duplicate removed

System.out.println(emails); // Output: [alice@example.com, bob@example.com]

2. What is an identity hashmap?

Identity HashMap is a special implementation of Map in Java that **compares keys using reference equality (==) instead of object equality (equals())**.

**Key Characteristics:**

* Unlike HashMap, which uses equals() and hashCode(), IdentityHashMap only considers **memory reference (identity comparison) using ==**.
* It is not thread-safe.
* It allows null keys and values.
* It is faster than HashMap in scenarios where reference-based comparison is required.

3.Hashmap implementation in java

Will Hear Durgesh lecture and write Notes

4.Importance of Hashcode and equals in Hashmap

Hear Lecture Understand Write here

5.Changes in HashMap in java 8 compare to java 7

| **Feature** | **Java 7 (HashMap)** | **Java 8 (HashMap)** |
| --- | --- | --- |
| **Collision Handling** | Used **Linked List** for storing multiple entries in the same bucket. | Uses **Balanced Tree (Red-Black Tree)** when collisions exceed **8** for better performance. |
| **Performance in High Collisions** | **O(n)** (due to linked list traversal). | **O(log n)** (due to Red-Black Tree). |
| **Rehashing Mechanism** | Used index = hashCode % capacity. | Used **improved hash function** to reduce collisions and better distribute keys. |
| **Computation of Index** | Directly used hashCode() and modulus. | Introduced **bitwise operations** ((n - 1) & hash) for faster computation. |
| **null Keys Handling** | Stored in **bucket index 0** (same as Java 8). | Same behavior, but better optimized due to hashing improvements. |
| **Thread-Safety** | Not thread-safe (requires external synchronization). | Not thread-safe but **optimized for concurrent access** (used in ConcurrentHashMap). |

 **Java 7:** When multiple keys have the same hash (collision), they are stored in a **Linked List** at the same bucket index. This results in **O(n) time complexity** in worst-case scenarios (if too many collisions occur).

 **Java 8:** If a bucket has more than **8 entries**, Java **converts the linked list into a Red-Black Tree**, reducing the search time to **O(log n)**.

6. Difference between LinkedList vs ArrayList

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| * ArrayList internally uses a **dynamic array** to store the elements. | * LinkedList internally uses a **doubly linked list** to store the elements. |
| * Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | * Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
|  |  |
| * ArrayList is **better for storing and accessing** data. | * LinkedList is **better for manipulating** data. |
| * The memory location for the elements of an ArrayList is contiguous. | * The location for the elements of a linked list is not contagious. |
| * Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | * There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized. |
|  |  |

7.HashSet implementation in java 8

HashSet is a **collection** in Java that implements the Set interface. It is backed by a HashMap and stores **unique elements** without maintaining any specific order.

**Key Features of HashSet**

✔ **No duplicate elements**  
✔ **No guaranteed order** (insertion order is not maintained)  
✔ **Allows one null element**  
✔ **Uses hashCode() and equals() for uniqueness**  
✔ **Underlying implementation: HashMap**

8. How to remove duplicate elements in a list using stream API

The easiest way to remove duplicates from a list using **Java 8 Stream API** is by using the distinct() method.

**Example:**

public class RemoveDuplicates {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5, 6, 6, 7);

// Remove duplicates

List<Integer> uniqueNumbers = numbers.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(uniqueNumbers); // Output: [1, 2, 3, 4, 5, 6, 7]

}

 Use **distinct()** if you want an easy, order-preserving approach.

 Use **toSet()** if you don't care about order and want better performance.

 Use **LinkedHashSet** if order matters.

9.Difference Between HashSet and TreeSet

Homogeneous data means **all elements are of the same type**.

int[] numbers = {1, 2, 3, 4, 5}; // All elements are integers

All elements are of type int - Ensures type safety

Only String objects are stored : Type safety is enforced

**Heterogeneous data means elements can be of different types.**

Object[] data = {1, "Hello", 3.14, true};

|  |  |  |
| --- | --- | --- |
| **Parameters** | **HashSet** | **TreeSet** |
| **Ordering or Sorting** | It does not provide a guarantee to sort the data. | It provides a guarantee to sort the data. The sorting depends on the supplied Comparator. |
| **Null Objects** | In HashSet, **only an element** can be null. | It does not allow null elements. |
| **Comparison** | It uses **hashCode()** or **equals()** method for comparison. | It uses **compare()** or **compareTo()** method for comparison. |
| **Performance** | It is **faster** than TreeSet. | It is **slower** in comparison to HashSet. |
| **Implementation** | Internally it uses **HashMap** to store its elements. | Internally it uses **TreeMap** to store its elements. |
| **Data Structure** | HashSet is backed up by a hash table. | TreeSet is backed up by a Red-black Tree. |
| **Values Stored** | It allows only **heterogeneous** value. | It allows only **homogeneous** value. |

10. collections in depth like how they work and implemented in java

In Java, the Collections Framework is a powerful architecture for storing and manipulating groups of objects, providing interfaces and classes for various data structures like lists, sets, and maps. It offers efficient implementations and algorithms, making it easier to work with collections of data.

Benefits of using the Collections Framework:

* **Reduced development time:** Provides ready-to-use data structures and algorithms.
* **Improved performance:** Offers efficient implementations of common data structures and algorithms.
* **Enhanced code readability:** Standardizes the way collections are handled, making code easier to understand and maintain.
* **Type safety:** Generics ensure that collections store only the specified type of objects, preventing runtime errors.

Collection (Interface)

│

├── List (Interface) ───> ArrayList, LinkedList, Vector, Stack

│

├── Set (Interface) ───> HashSet, LinkedHashSet, TreeSet

│

├── Queue (Interface) ───> PriorityQueue, Deque (LinkedList, ArrayDeque)

│

└── Map (Interface) ───> HashMap, LinkedHashMap, TreeMap, Hashtable, ConcurrentHashMap

11.what is self balancing tree

A self-balancing binary search tree (BST) is a data structure that automatically maintains a balanced structure, ensuring optimal performance for operations like searching, inserting, and deleting,

A self-balancing BST is a type of binary search tree that dynamically adjusts its structure after each insertion or deletion to prevent it from becoming skewed or degenerate (like a linked list).

12. Array vs ArrayList diff?

In Java, arrays are fixed in size, meaning once they are created, you cannot change their size, whereas ArrayLists are dynamic, allowing elements to be added or removed after initialization.

Arrays can store both primitive data types (like int, char) and objects, while ArrayLists can only store objects and use generics for type-safety.

Arrays are not implemented on some standard data structure hence readymade methods are not available ArrayList Implemented on standard data structure hence readymade method Support is available

13. Map (Different classes, Iteration method)

In Java, a Map is an interface in the Java Collections Framework, part of the java.util package. It represents a collection of key-value pairs, where each key is associated with exactly one value. Some common implementations of Map are HashMap, TreeMap, and LinkedHashMap.

There are different ways to iterate over a Map in Java, depending on your needs.

HashMap is one of the most commonly used Map implementations. It allows null values and keys but doesn't guarantee any specific order of the elements.

Map<String, Integer> map = new HashMap<>();

map.put("Apple", 1);

map.put("Banana", 2);

map.put("Cherry", 3);

for (Map.Entry<String, Integer> entry : map.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

for (String key : map.keySet()) {

System.out.println(key + ": " + map.get(key));

}

for (Integer value : map.values()) {

System.out.println(value);

}

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

 **entrySet()**: Gives both the key and value as a pair, which is very efficient for iteration.

 **keySet()**: Iterates only over the keys, and you must call map.get(key) to get the value.

 **values()**: Iterates only over the values, which is useful if you don't need the keys.

 **Lambda with forEach()**: A concise way to iterate using Java 8+ features.

14. what happen when we put same key with different value in hashmap

When you insert a new entry into a HashMap with a key that already exists, the new value will **overwrite** the previous value associated with that key. The HashMap does not allow duplicate keys, so only the most recent value will be stored for that key.

 **Overwriting**: When you put a key with a new value that already exists in the HashMap, the old value is replaced with the new one.

 **No Duplicate Keys**: A HashMap does not allow duplicate keys. Even if you add a key multiple times with different values, only the last inserted value will be retained for that key.

15. print the hashmap in accending order of keys

Since a HashMap **does not maintain any order**, you can use a **TreeMap** to store and sort the entries by keys automatically.

A **TreeMap** automatically sorts keys in **natural order** (ascending).

public static void main(String[] args) {

// Creating a HashMap

HashMap<Integer, String> map = new HashMap<>();

map.put(3, "C");

map.put(1, "A");

map.put(4, "D");

map.put(2, "B");

// Sorting by converting HashMap to TreeMap

TreeMap<Integer, String> sortedMap = new TreeMap<>(map);

// Printing in ascending order of keys

for (Map.Entry<Integer, String> entry : sortedMap.entrySet()) {

System.out.println(entry.getKey() + " -> " + entry.getValue());

}

16. how can we create concurrent hashmap

A **ConcurrentHashMap** is a thread-safe implementation of HashMap that allows **multiple threads to read and write concurrently** **without locking the entire map**.

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

17. give me more details on iterator

An **Iterator** is an interface in Java that allows **traversing a collection** (like List, Set, or Map) **sequentially**. It provides a way to iterate over elements **without exposing underlying structure**.

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) { // Check if elements are available

String fruit = iterator.next(); // Get the next element

System.out.println(fruit);

18.What is Treemap

The TreeMap provides a sorted order of its elements, based on the natural order of its keys or a custom Comparator passed to the constructor. This makes it useful in situations where you need to retrieve elements in a specific order

 **Key-value pairs**: It maps each key to a value, just like any other Map implementation.

 **No duplicate keys**: Just like HashMap, TreeMap does not allow duplicate keys. If you insert a key-value pair with an existing key, the old value is replaced by the new value.

19. which one is better hashmap concurrent hashmap – Concurrent Hash Map

 **Use HashMap** when:

* You are working in a **single-threaded** or **low-concurrency** environment.
* You don’t need thread safety and are confident there will be no concurrent modifications.
* You need to allow **null keys and values**.

 **Use ConcurrentHashMap** when:

* You are working in a **multi-threaded environment** where multiple threads will be concurrently accessing and modifying the map.
* You need **high performance** with thread safety and you can’t afford locking the entire map.
* You **cannot allow null keys or values** (since ConcurrentHashMap does not permit them).

20. Does hasmap can have null value?

Yes, a **HashMap** in Java **can** have null values. Java's HashMap<K, V> allows:

1. **Null values**: You can store null as a value in a HashMap.
2. **A single null key**: A HashMap allows at most one null key.

21. Entry set what is usage

The entrySet() method in HashMap returns a **Set** of key-value pairs (Map.Entry<K, V>) from the HashMap. This is useful when you need to iterate over both **keys and values** at the same time.

map.put(3, "Cherry");

for (Map.Entry<Integer, String> entry : map.entrySet())

System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());

Exception Handling In Java

1.Exception handling, will finally execute if try throws exception (without a catch) ?

Yes, in Java, the finally block will execute even if an exception is thrown in the try block, even if there is no catch block to handle that exception. The finally block is guaranteed to run, regardless of whether an exception was thrown or not. This ensures that certain cleanup code (like closing resources) is always executed.

When finally Does NOT Execute -- If System.exit(0) is called in try

2. Diff between checked and unchecked exception

In Java, exceptions are categorized into **checked** and **unchecked** exceptions based on whether they are **checked at compile-time** or **runtime**.

**Checked Exceptions**

✔ **Checked at compile-time**  
✔ Must be **handled using try-catch or declared using throws**  
✔ Typically represent **recoverable conditions** (e.g., file not found, database issues)

**Examples of Checked Exceptions :** IOException**,**FileNotFoundException**,**InterruptedException

**Example:**

public class Main {

public static void main(String[] args) {

try {

FileReader file = new FileReader("file.txt"); // Checked Exception

} catch (FileNotFoundException e) {

System.out.println("File not found: " + e.getMessage());

}

🔹 **Without handling (try-catch or throws), the compiler gives an error.**

**Unchecked Exceptions**

✔ **Checked at runtime** (not at compile-time)  
✔ **No requirement to handle them explicitly**  
✔ Typically result from **programming logic errors** (e.g., null pointer, array index out of bounds)

**Examples of Unchecked Exceptions :** NullPointerException,ArithmeticException**,** ArrayIndexOutOfBoundsException**,**IllegalArgumentException

**Example: Unchecked Exception**

public class Main {

public static void main(String[] args) {

int num = 10 / 0; // ArithmeticException (Unchecked)

System.out.println("This line will not execute.");

}

}

🔹 **Unchecked exceptions occur at runtime and crash the program if not handled.**

3.Can we throw an unchecked exception in child class if parent class doesn’t throw any exception

**Yes, a child class can throw an unchecked exception even if the parent class does not declare any exceptions.**

This is because **unchecked exceptions (RuntimeException and its subclasses)** are **not checked at compile-time**, so there are no restrictions on them.

❌ **A child class cannot throw new checked exceptions if the parent class does not declare them.**  
If you try to do so, the compiler will throw an error.

4. What is try with resource in java

**Try-With-Resources** is a feature in Java (introduced in **Java 7**) that **automatically closes resources** (like files, sockets, database connections) **after execution**.

✅ **No need for an explicit finally block to close resources.**  
✅ **Prevents resource leaks** by ensuring resources are closed properly.

5. What is the significance of finally block

The **finally block** in Java is used to **execute important cleanup code** (like closing files, database connections, or releasing resources) **regardless of whether an exception occurs or not**.

✅ **Always Executes** (Even if an exception occurs in try block)  
✅ **Ensures Resource Cleanup** (Files, DB connections, sockets, etc.)  
✅ **Executes Even if return is Called** in try or catch  
✅ **Does Not Execute Only If JVM Terminates (System.exit(0))**

**6.** Can we have try block without catch block

Yes, in Java, you can have a **try block** without a **catch block**, but it must be followed by a **finally block**. This is known as a **try-with-finally** block.

7. Diff between error and exception

| **Feature** | **Error** | **Exception** |
| --- | --- | --- |
| **Definition** | Represents serious problems that **a program cannot handle** | Represents issues that **a program can handle and recover from** |
| **Caused By** | Issues **beyond application control** (JVM failures, system crashes, memory leaks) | **Programming mistakes** (invalid input, null references, arithmetic errors) |
| **Handled using try-catch?** | ❌ **No** (Should not be caught in most cases) | ✅ **Yes** (Can be caught using try-catch) |
| **Can program recover?** | ❌ **No**, usually fatal | ✅ **Yes**, if handled properly |
| **Example Classes** | OutOfMemoryError, StackOverflowError, VirtualMachineError | NullPointerException, IOException, ArithmeticException |

8.when and in what situation you use exception handling and throws

✔ Use **try-catch** when you **want to handle an exception immediately** and ensure program continuity.  
✔ Use **throws** when a method **doesn’t know how to handle an exception** and wants the **caller to manage it**.  
✔ Use try-catch for **application-level** exceptions (like invalid input, division by zero).  
✔ Use throws for **framework-level** exceptions (like JDBC, API development).

9. parrent class of all exception

✔ **Throwable** is the **base class** for all **errors** and **exceptions** in Java.  
✔ **Exception** handles **recoverable errors**.  
✔ **Error** represents **serious issues** that should not be caught.

java.lang.Object

└── java.lang.Throwable 👈 (Parent class of all exceptions & errors)

├── java.lang.Exception 👈 (Checked & Unchecked Exceptions)

│ ├── java.io.IOException

│ ├── java.sql.SQLException

│ ├── java.lang.RuntimeException 👈 (Unchecked exceptions)

│ ├── java.lang.NullPointerException

│ ├── java.lang.ArithmeticException

│ ├── java.lang.ArrayIndexOutOfBoundsException

│

└── java.lang.Error 👈 (Serious system-level issues)

├── java.lang.OutOfMemoryError

├── java.lang.StackOverflowError

10. child classes of throwable

The **Throwable** class is the **root class** of all exceptions and errors in Java. It has two direct child classes:

1. **Exception** → Represents **recoverable errors** (should be handled).
2. **Error** → Represents **serious system failures** (should not be caught).

Object oriented Programming System Interview Questions

1. final keyword in class, method and variable

✔ **final Variable** → Value cannot change after initialization.  
✔ **final Method** → Cannot be overridden in subclasses.  
✔ **final Class** → Cannot be inherited.

2. Why is String immutable

✔ Once a String object is created, its **contents cannot be modified**.  
✔ Any operation that seems to modify a String actually **creates a new String object**.

✔ **Java uses a special memory area called the String Pool**.  
✔ When a String is created, it is **stored in the pool**.  
✔ If a String is already in the pool, the JVM **reuses the existing object instead of creating a new one**.  
✔ Immutability prevents changes to existing strings, making pooling efficient.

✔ String is immutable for performance, security, and efficiency**.**

**3.** will you store password in string or char array

Use char[] instead of String to store passwords in Java.

Even after reassignment, the old password stays in memory until GC removes it.

**char[] is mutable** → You can **explicitly erase** it after use. **Not stored in String Pool** → No risk of sensitive data being stored indefinitely.

4. Can we override static method? -- No, static methods cannot be overridden in Java!

✔ **Static methods belong to the class, not an instance (object).**  
✔ **Method overriding is based on runtime polymorphism (dynamic binding), but static methods use compile-time binding (static binding).**

**5.** What is method reference

**Method reference** is a **shorthand** notation of writing **lambda expressions** to refer to an existing method.  
🔹 It **eliminates boilerplate code** and makes the code more readable.  
🔹 **Introduced in Java 8** to improve functional programming.

6. What is autoboxing

**Autoboxing** is the automatic conversion of **primitive types** into their **corresponding wrapper class objects** when needed.

🔹 **Autoboxing** helps in **converting primitives to wrapper objects automatically**.  
🔹 **Unboxing** does the **reverse (wrapper to primitive)**.

7. What is singleton class in java and how to implement it

In Java, a singleton class ensures only one instance of itself exists throughout the application's lifecycle, providing a global point of access to that instance. You achieve this by making the constructor private, creating a static instance variable, and providing a public static method to access it.

8. Why we need singleton class?

A **singleton class** ensures that **only one instance** of a class exists throughout the application's lifecycle. This helps in **resource management, performance optimization, and global access**.

🔹 Instead of creating multiple instances, a singleton **reuses a single instance**.  
🔹 **Example:** **Database Connection Pooling** → Creating multiple connections is expensive; Singleton ensures a single connection manager is used.

9. How to make a singleton class thread safe?

Yes Using synchronized Method and Double-Checked Locking Bill Pugh Singleton

10. Why java doesn’t support multiple inheritance

🔹 **Multiple inheritance** means a class can inherit **from more than one parent class**.

**To Avoid the Diamond Problem**

🔹 Multiple inheritance makes **code harder to read, debug, and maintain**.  
🔹 Java follows the **principle of simplicity** → Keeping a **clear, single inheritance path**.

**✔ Java supports multiple inheritance through interfaces.  
✔ Interfaces only define methods (without implementation), so there is no ambiguity.**

11. Can we override a private method

**No, we cannot override a private method in Java. Why?  
✔ Private methods are not inherited by child classes, so they are not accessible outside their defining class.  
✔ Method overriding requires inheritance, but since private methods are not inherited, they cannot be overridden.**

**12.** Diff b/w static variable vs Instance Variable

1**.**Static Variable (static) → Shared among all objects.  
2️.Instance Variable → Unique for each object.  
3️.Memory Usage → Static variables use less memory (stored once), while instance variables take more memory (stored per object).  
4️.Modification Impact → Changing a static variable affects all objects, but changing an instance variable affects only that object.

static int companyRevenue = 100000; // Static variable (shared by all employees)

int salary; // Instance variable (unique for each employee)

**✔ Static variable (companyRevenue) is shared, while salary is different for each object.**

13. class A {object show() } class B ext class A {string show()} class c ext B {integer show()} --Will this work

Your code will **not work** because **method overriding does not allow changing the return type arbitrarily**.

🔹 **Covariant return type** allows **only subclasses** of the original return type.  
🔹 You are changing the return type in each subclass **to an unrelated type** (Object → String → Integer), which **violates the overriding rule**.

✅ **Covariant return types must be subclasses of the original return type.**  
✅ String is a subclass of Object, so **String show()** in B is fine.  
❌ Integer is **not** a subclass of String, so **Integer show()** in C is **not valid**.

14. for financial systems which data type should be used to maintain the precision (i said BigDecimal)

BigDecimal provides **arbitrary-precision arithmetic**, ensuring **accurate calculations** for money and transactions.

 In **banking applications**, even **a small precision error** can **accumulate**, causing huge discrepancies in large transactions.

 BigDecimal eliminates such rounding issues.

15. How to use Multiple inheritance ?

🔹 **Java does NOT support multiple inheritance with classes** to avoid the **diamond problem** (ambiguity in method resolution).  
🔹 However, **multiple inheritance is possible** using **interfaces**.

16. String builder and buffe Diff

StringBuffer is synchronized, meaning its methods are thread-safe and can be safely used in a multithreaded environment.

StringBuilder is not synchronized, which makes it faster than StringBuffer, but it is not thread-safe and should not be used in a multithreaded environment

Use String Buffer for Multithreading Environment

17. Method OverLoading vs Method OverRiding

Two or More methods having Same method name, different parameters is Known as Method Overloding

Overloading is a **compile-time polymorphism** (also known as static polymorphism).

The method in the subclass should have the **same name, same return type, and same parameters** as the method in the superclass. Is called method oriding

Overriding is a **runtime polymorphism** (also known as dynamic polymorphism).

18. Static keyword in Java, can we Overload/Override Static method

✔ **Static Methods Can Be Overloaded** (same name, different parameters).  
❌ **Static Methods Cannot Be Overridden** (they are hidden instead).  
✔ **Overriding works only with instance methods, not static ones.**

19. Interface and abstract class

The **abstract keyword** is used to declare abstract class. The **interface keyword** is used to declare interface.

Abstract class **doesn't support multiple inheritance**. Interface **supports multiple inheritance**.

The abstract class does not have body the is provided by Extended class abstract class cannot be used to create object to access it must be inherited from another classs

20. How to use Multiple inheritance ?

Java Doesn’t not support multiple inheritance to avoid the **diamond problem**. To achieve multiple inheritance using Interface because interface we don’t have mention implementation

However, **it allows multiple inheritance through interfaces** because **interfaces only define method signatures**

21. do you think java is completely object oriented

No, Java is not a purely object-oriented language because it supports primitive data types (like int, float, boolean) which are not objects, and it allows static members, which are not associated with any particular object.

22. reasons why java is not completely object oriented

 Java **restricts multiple inheritance with classes** to avoid ambiguity (diamond problem).

 A purely OOP language **would allow multiple class inheritance**.

it supports primitive data types (like int, float, boolean) which are not objects,

23. what use of static variable what should be the scinario of using static variable

If you want **a single shared value** among all instances of a class, use a static variable.

✔ If a value **should never change** and is shared across all instances, use static final.

✔ If multiple objects need access to the **same database connection**, store it in a static variable.

24. Why main is static method

✔ If main() were **not static**, the JVM would need to create an object before calling it.  
✔ But how can the JVM create an object **if main() itself is the entry point?** ❌ **It would be a deadlock!**

✔ If main() were **non-static**, a **new object** would be created every time the program starts.

✔ Since main() is **static**, the JVM can **directly call it** without creating an instance.

25. how can we make constructor static

No, **constructors cannot be static** in Java. If you try to declare a constructor as static, the compiler will throw an error. Constructor Is Meant for Object Creation

26. difference between == and .equals()

✔ The == operator **compares memory addresses (references), not values**.  
✔ It checks if two variables **point to the same object in memory**.

✔ The .equals() method is used to **compare the actual content (values) of objects**.

27. usage of string constant pool

✔ The **String Constant Pool** (also known as **String Intern Pool**) is a special memory area inside the **Heap** that **stores String literals to save memory** and **improve performance**.  
✔ When you create a String literal (String s = "Hello";), Java **first checks if the same value already exists in the pool**. If it does, it **reuses the existing object** instead of creating a new one.

public class StringPoolExample {

public static void main(String[] args) {

String s1 = "Hello";

String s2 = "Hello";

String s3 = new String("Hello");

String s4 = new String("Hello");

System.out.println(s1 == s2); // ✅ true (Same object from pool)

System.out.println(s1 == s3); // ❌ false (new String creates a new object)

✔ s1 and s2 **point to the same object** in the **String Pool**, so s1 == s2 returns true.  
✔ s3 is created using new String(), so it’s **not stored in the pool**, making s1 == s3 return false.

Create with String Literal ("Hello") Stores in String Constant pool memory reused

Create new String("Hello") Stores in Heap memory it creates a new object

┌───────────────────────────────┐

│ \*\*String Constant Pool\*\* │

│ ┌───────────────────────────┐ │

s1 ─────────────▶│ │ "Hello" │ │

s2 ─────────────▶│ └───────────────────────────┘ │

└───────────────────────────────┘

▲

│ (Reused)

│

▼

┌───────────────────────────────────────────────────┐

│ \*\*Heap Memory\*\* │

│ ┌─────────────────────────┐ ┌─────────────────┐ │

s3 ─▶ │ new String("Hello") │ │ new String("Hello") │ ◀─ s4

│ (Separate object, not in │ │ (Separate object) │ │

│ String Pool) │ │ │ │

└───────────────────────────┘ └───────────────────┘ │

└───────────────────────────────────────────────────┘

28. what are all immutable classes

In Java, immutable classes, whose objects cannot be modified after creation, include String, all wrapper classes (Integer, Double, Boolean, etc.), and classes like LocalDate, LocalTime, UUID, and BigDecimal.

29. how we can create immutable class

✔ **Declare the class as final** → Prevent subclassing (which could break immutability).  
✔ **Make all fields private final** → Ensure fields cannot be modified after assignment.  
✔ **Do not provide setter methods** → Prevent changing values after object creation.  
✔ **Initialize fields only through a constructor** → Ensure values are assigned only once.

30. why should you declare immutable class as final

Declaring an **immutable class as final** **prevents subclassing**, which is crucial because **subclasses can break immutability** by overriding methods and modifying the object's state.

Coding Questions

program to reverse string in java

public class ReverseString {

public static void main(String[] args) {

String str = "Hello";

String reversed = new StringBuilder(str).reverse().toString();

System.out.println("Reversed String: " + reversed);

}

}

Using for loop

public class ReverseStringLoop {

public static void main(String[] args) {

String str = "Hello";

String reversed = "";

for (int i = str.length() - 1; i >= 0; i--) {

reversed += str.charAt(i);

}

System.out.println("Reversed String: " + reversed);

}

}

How to reverse array list

public class ReverseArrayList {

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));

Collections.reverse(list); // 🔄 Reverses the list in-place

System.out.println("Reversed List: " + list);

}

write the logic for string count all characters occurrence programme

public class CharacterFrequency {

public static void main(String[] args) {

String str = "hello world";

Map<Character, Integer> frequencyMap = new HashMap<>();

// Loop through each character in the string

for (char c : str.toCharArray()) {

frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);

}

System.out.println("Character Frequency: " + frequencyMap);

}

}

public class SpecificCharacterCount {

public static void main(String[] args) {

String str = "hello world";

char target = 'o'; // Character to count

int count = 0;

// Loop through each character in the string

for (char c : str.toCharArray()) {

if (c == target) {

count++;

}

}

System.out.println("Character '" + target + "' appears " + count + " times.");

}

}

write the logic for Remove Array duplicates without collection

java8 stream to print sum of n number

import java.util.stream.IntStream;

public class SumOfNumbers {

public static void main(String[] args) {

int n = 10; // Let's sum the first 10 numbers

// Using Java 8 Streams to calculate the sum

int sum = IntStream.range(1, n + 1) // Create a stream of numbers from 1 to n

.sum(); // Calculate the sum

// Print the result

System.out.println("The sum of the first " + n + " numbers is: " + sum);

}

}

The sum of the first 10 numbers is: 55

java8 stream syntax to print words and frequency of an array

import java.util.Arrays;

import java.util.Map;

import java.util.stream.Collectors;

public class WordFrequency {

public static void main(String[] args) {

String[] words = {"apple", "banana", "apple", "orange", "banana", "apple"};

// Using Stream to count word frequency

Map<String, Long> wordCount = Arrays.stream(words)

.collect(Collectors.groupingBy(word -> word, Collectors.counting()));

// Print word frequency

wordCount.forEach((word, count) -> System.out.println(word + " -> " + count));

}

}