1. Tell me about your project and roles and responsibilities in detail. Questions related to

project architecture, etc.

2. What are the collections you know?

3. List disadvantages.

 **Performance Overhead**: Collections can introduce performance overhead due to the abstraction and additional functionality they provide, which can be slower than using arrays for simple data storage.

 **Memory Consumption**: Collections can consume more memory than primitive arrays because they require additional object overhead for their internal structures.

 **Complexity**: The variety of collection types (e.g., List, Set, Map) and their specific behaviors can add complexity to code, making it harder to understand and maintain.

 **Type Safety**: Although generics help with type safety, using raw types can lead to runtime ClassCastException. Careful use of generics is necessary to avoid this.

 **Synchronization Issues**: The default collections are not synchronized, which can lead to issues in multithreaded environments unless explicitly handled (e.g., using Collections.synchronizedList()).

 **Iteration Performance**: Depending on the type of collection, iteration can be slower than using arrays. For example, accessing elements in a LinkedList is generally slower than in an array or ArrayList.

 **Limited Primitive Support**: Collections do not directly support primitive types. Instead, they rely on wrapper classes (e.g., Integer, Double), which can introduce boxing and unboxing overhead.

 **Inflexibility**: Some collections have fixed sizes or constraints that limit their flexibility. For example, ArrayList can grow, but Arrays.asList() creates a fixed-size list.

 **Default Implementations**: Some collection implementations may not be optimal for specific use cases. For instance, using a HashSet may not be ideal if ordering is essential.

 **Learning Curve**: For beginners, understanding the differences between various collection types and their use cases can be challenging.

 **Garbage Collection**: Heavy use of collections can lead to increased garbage collection overhead, especially if large numbers of temporary collections are created and discarded.

 **Complex Sorting**: Sorting custom objects in collections may require implementing comparators, which can add complexity to the code.

4. Map internal working. Questions on Key as an Employee object: what will happen if

you don’t implement equals and hash code, what if only implements equals, hash

code?

 **Buckets**:

* Internally, a HashMap uses an array of buckets (called an array of linked lists or trees) to store the key-value pairs. Each bucket corresponds to a hash code value.

 **Hash Function**:

* When you insert a key-value pair, the key is processed by a hash function to compute an integer hash code. This hash code determines the index of the bucket in the array where the entry will be stored.
* The hash function distributes keys uniformly across the buckets to minimize collisions.

 **Collisions**:

* If two different keys generate the same hash code (collision), they will end up in the same bucket.
* In HashMap, collisions are handled using a linked list (or a balanced tree for large buckets, as of Java 8):
  + **Linked List**: If multiple keys hash to the same bucket, they are stored in a linked list.
  + **Tree**: If the number of entries in a bucket exceeds a certain threshold (usually 8), the linked list is transformed into a balanced tree (e.g., a red-black tree) to improve lookup time.

 **Load Factor and Rehashing**:

* The load factor is a measure of how full the map can get before it needs to increase its capacity. The default load factor is 0.75, which means the HashMap will resize (rehash) when it is 75% full.
* When rehashing occurs, a new array of buckets is created, and all existing entries are rehashed and moved to the new array.

5. How does Map work?

In Java, the Map interface is part of the Java Collections Framework, and it provides a way to store and manage key-value pairs. Each key in a Map is unique, and each key maps to exactly one value. Here’s an overview of how Map works in Java, focusing on its key implementations like HashMap, TreeMap, and LinkedHashMap.

**Key Components of Map**

1. **Key-Value Pair**:
   * A Map is a collection of entries, where each entry consists of a key and its associated value. The key is used to retrieve the value.
2. **Unique Keys**:
   * In a Map, each key must be unique. If you try to add a key that already exists, the new value will overwrite the existing value.
3. **Null Keys and Values**:
   * HashMap allows one null key and multiple null values.
   * TreeMap does not allow null keys (it uses natural ordering or a comparator to sort keys).
   * LinkedHashMap allows one null key and multiple null values, like HashMap.

**Common Implementations**

1. **HashMap**:
   * **Storage**: Uses an array of buckets to store entries. Each bucket can contain multiple entries (using linked lists or trees for collision handling).
   * **Performance**: Average time complexity for operations like put, get, and remove is O(1). Worst-case time complexity can degrade to O(n) in scenarios with many hash collisions.
   * **Order**: Does not maintain any order of keys.
2. **TreeMap**:
   * **Storage**: Implements a Red-Black tree, a self-balancing binary search tree.
   * **Performance**: Operations like put, get, and remove have a time complexity of O(log n).
   * **Order**: Maintains a sorted order of keys based on their natural ordering or a specified comparator.
3. **LinkedHashMap**:
   * **Storage**: Combines the functionality of HashMap with a linked list to maintain insertion order.
   * **Performance**: Similar to HashMap, with average time complexity for basic operations as O(1).
   * **Order**: Maintains the order in which keys were added (insertion order).

6. Question: Is this class immutable? If yes, how? If no, how to make it an immutable

class?

public class Immutable {

List list;

}

7. Create an immutable class using a list.

a) make class final

b) make properties final

c) don’t create setter methods

// Constructor

public ImmutableListExample(List<String> items) {

// Create a defensive copy of the input list to prevent external modification

this.items = Collections.unmodifiableList(List.copyOf(items));

}

8. So, HashMap will work on equals(), but I want to use ==. Which collection will you

use? Ans: Identity HashMap; it stores key reference.

9. What is the difference between inheritance and encapsulation? Questions related to

the topic?

10. Can we do inheritance in the parent class without extends?

Through interface we can achieve.

11. Polymorphism? Method overloading and method overriding.

12. Talk something about serialization and deserialization? What is the use? Can’t we do

serialization without implementing the Serializable interface?

Serialization in Java is the process of converting an object into a byte stream, which can then be persisted to a file or sent over a network. Deserialization is the reverse process, where the byte stream is converted back into a copy of the original object.

Java provides built-in support for serialization through the java.io.Serializable interface. Below, I'll explain how to implement serialization and deserialization step by step.

Steps for Serialization and Deserialization

Implement the Serializable Interface

Create Serialization Method

Create Deserialization Method

Example Code

Here's a complete example demonstrating serialization and deserialization in Java.

Step 1: Implement the Serializable Interface

Create a class that implements Serializable. This class can have any fields you want to serialize.

java

Copy code

import java.io.Serializable;

class Person implements Serializable {

private static final long serialVersionUID = 1L; // Recommended for Serializable classes

private String name;

private int age;

// Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getters

public String getName() {

return name;

}

public int getAge() {

return age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + '}';

}

}

The serialVersionUID is a unique identifier for Serializable classes. It’s a good practice to declare this field to ensure version control during the serialization process.

Step 2: Create Serialization Method

This method will serialize the Person object and write it to a file.

java

Copy code

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectOutputStream;

public class SerializationExample {

public static void serializePerson(Person person, String filePath) {

try (FileOutputStream fileOut = new FileOutputStream(filePath);

ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

out.writeObject(person);

System.out.println("Serialized data is saved in " + filePath);

} catch (IOException e) {

e.printStackTrace();

}

}

}

Step 3: Create Deserialization Method

This method will read the serialized Person object from a file and return it.

java

Copy code

import java.io.FileInputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

public class DeserializationExample {

public static Person deserializePerson(String filePath) {

Person person = null;

try (FileInputStream fileIn = new FileInputStream(filePath);

ObjectInputStream in = new ObjectInputStream(fileIn)) {

person = (Person) in.readObject();

System.out.println("Deserialized Person: " + person);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

return person;

}

}

13. ArrayList vs LinkedList? How both will work internally? Can you explain in detail?

Some questions related to Time Complexity.

16. Fibonacci series using a recursive method.

17. Swap two numbers without using a third variable.

18. How does ArrayList result in faster data retrieval internally?

ArrayList in Java is often considered faster for data retrieval compared to other collection types, such as LinkedList, due to its underlying data structure and the way it handles indexing. Here's a detailed explanation of why ArrayList is typically faster for data retrieval:

### 1. ****Underlying Data Structure****

* **Array-Based**: ArrayList is backed by a dynamic array. This means that elements are stored in contiguous memory locations. The elements in an ArrayList can be accessed directly via their index, which allows for very fast retrieval.

### 2. ****Constant Time Complexity****

* **Direct Index Access**: Accessing an element by its index in an ArrayList is an O(1) operation. This means that the time taken to retrieve an element does not depend on the size of the list.

java

Copy code

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

list.add("A");

list.add("B");

list.add("C");

String value = list.get(1); // Fast access, O(1)

### 3. ****Cache Locality****

* **Contiguous Memory**: Since the elements in an ArrayList are stored in contiguous memory locations, they take advantage of CPU caching. When an element is accessed, the adjacent elements are likely loaded into the cache, making subsequent accesses faster.

### 4. ****Fewer Overheads****

* **No Node Overhead**: In contrast, a LinkedList stores elements in nodes, each of which contains references (pointers) to the next and previous nodes. This means that, in addition to the data, there are extra memory and processing overheads associated with managing those references.

19. Time complexity of adding elements in LinkedList at the beginning and ending and

how it is different internally.

### Time Complexity

1. **Adding at the Beginning**:
   * **Operation**: addFirst() or add(0, element)
   * **Time Complexity**: O(1)
   * **Explanation**: When adding an element at the beginning of a LinkedList, you simply create a new node and point its next reference to the current head of the list. Then you update the head to this new node. This operation does not require traversing the list, making it a constant time operation.
2. **Adding at the End**:
   * **Operation**: addLast() or add(element) (if the last node is not directly referenced)
   * **Time Complexity**:
     + **O(1)** if you maintain a tail reference.
     + **O(n)** if you do not maintain a tail reference and need to traverse the list to find the last element.

20. How does ConcurrentHashMap work internally?

### Internal Structure

Here's a simplified representation of the internal structure of ConcurrentHashMap:

1. **Array of Segments**:
   * The map is backed by an array of segments. Each segment is essentially a HashMap or a similar structure.
2. **Segment Locks**:
   * Each segment has its own lock (a ReentrantLock), which is used to control access to the entries within that segment.
3. **Hashing and Buckets**:
   * Each segment uses its own hashing mechanism and bucket array to store the entries. When you put an entry into the map, the hash code is calculated, and the entry is placed in the appropriate segment based on that hash code.

21. Have you created an immutable class? What are the rules for creating an immutable

class?

22. Can we have a setter in the immutable class or constructor only?

23. What is the output?

public class Test {

public void methodTest(Object object) {

System.out.println("Calling object method");

}

public void methodTest(String object) {

System.out.println("Calling String method");

}

public static void main(String args[]) {

Test moe=new Test();

moe.methodTest(null);

}

}

24. What is the output?

public class ExceptionTest {

public static void main(String[] args) {

System.out.println(exceptionTest());

}

public static int exceptionTest() {

int i=6;

try {

throw new NullPointerException();

}

catch(NullPointerException e) {

i=10;

throw e;

}

finally {

i=20;

System.out.println("In finally block");

return i;

}

}

}

25. We have a string with duplicate characters, can you explain logic to find duplicate

character occurrence?

26. Can you write an expression to find the employee age > 20?

27. What is polymorphism and its types?

28. Why can't we reduce overridden method visibility?