**Assignment Collection Framework and Maps**

Q1. What is the Collection framework in Java

**The Collection framework in Java is a unified architecture for storing and manipulating groups of objects. It includes:**

1. **Interfaces: Define the abstract data types (e.g., List, Set, Queue, Map).**
2. **Implementations: Concrete classes like ArrayList, HashSet, LinkedList, and HashMap that implement these interfaces.**
3. **Algorithms: Methods for manipulating collections, like sorting and searching (e.g., Collections.sort()).**

**It helps in managing data more efficiently and is part of the java.util package.**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) 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. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |
| 5) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |
| 6) 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) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list interface. |

Q2. What is the difference between ArrayList and LinkedList

Q3. What is the difference between Iterator and ListIterator

| **Iterator** | **ListIterator** |
| --- | --- |
| Can traverse elements present in Collection only in the forward direction. | Can traverse elements present in Collection both in forward and backward directions. |
| Helps to traverse Map, List and Set. | Can only traverse List and not the other two. |
| Indexes cannot be obtained by using Iterator. | It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List. |
| Cannot modify or replace elements present in Collection | We can modify or replace elements with the help of set(E e) |
| Cannot add elements and it throws ConcurrentModificationException. | Can easily add elements to a collection at any time. |
| Certain methods of Iterator are next(), remove() and hasNext(). | Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e). |

Q4.What is the difference between Iterator and Enumeration

| **Feature** | **Iterator** | **Enumeration** |
| --- | --- | --- |
| **Supported Collections** | Universal, applicable to all collection classes in the Java Collections Framework. | Limited to legacy classes like Vector and Hashtable. |
| **Operations Supported** | Supports read and remove operations. | Only supports read operation. |
| **Methods** | hasNext(), next(), remove(). | hasMoreElements(), nextElement(). |
| **Fail-Fast Behavior** | Provides fail-fast behavior during iteration. | Does not support fail-fast behavior. |
| **Usage in Modern Collections** | Preferred for traversing modern collections like HashMap, ArrayList, etc. | Obsolete in modern applications; primarily used for legacy collections. |

Q5. What is the difference between List and Set

| **List** | **Set** |
| --- | --- |
| 1. The List is an indexed sequence. | 1. The Set is an non-indexed sequence. |
| 2. List allows duplicate elements | 2. Set doesn’t allow duplicate elements. |
| 3. Elements by their position can be accessed. | 3. Position access to elements is not allowed. |
| 4. Multiple null elements can be stored. | 4. Null element can store only once. |
| 5. List implementations are ArrayList, LinkedList, Vector, Stack | 5. Set implementations are HashSet, LinkedHashSet. |

Q6. What is the difference between HashSet and TreeSet

| **Sr. No.** | **Key** | **Hash Set** | **Tree Set** |
| --- | --- | --- | --- |
| 1 | Implementation | Hash set is implemented using HashTable | The tree set is implemented using a tree structure. |
| 2 | Null Object | HashSet allows a null object | The tree set does not allow the null object. It throws the null pointer exception. |
| 3 | Methods | Hash set use equals method to compare two objects | Tree set use compare method for comparing two objects. |
| 4 | Heterogeneous object | Hash set doesn't now allow a heterogeneous object | Tree set allows a heterogeneous object |
| 5 | Ordering | HashSet does not maintain any order | TreeSet maintains an object in sorted order |

Q7. What is the difference between Array and ArrayList

|  |  |
| --- | --- |
| **Array** | **ArrayList** |
| Fixed size | Dynamic size |
| Value type elements | Reference type elements |
| Can be multidimensional | Single-dimensional |
| Type-safe | Not type-safe |
| A bit faster than ArrayList | A bit slower than Array |

Q8. What is a Map in Java

In Java, a **Map** is an interface in the Collection framework that stores **key-value pairs**. It allows you to associate a specific key with a value, where each key is unique, but values can be duplicated. Common implementations of the Map interface include:

* **HashMap**: Unordered map with fast access.
* **TreeMap**: Sorted map based on keys.
* **LinkedHashMap**: Maintains insertion order.

Maps are used when you need to look up values based on unique keys efficiently.

Q9. What are the commonly used implementations of Map in Java

The commonly used implementations of the **Map** interface in Java are:

1. **HashMap**:
   * Stores key-value pairs in no particular order.
   * Allows one null key and multiple null values.
   * Provides fast performance for insertions, deletions, and lookups (constant time complexity on average).
2. **LinkedHashMap**:
   * Maintains the insertion order of keys.
   * Slightly slower than HashMap due to its linked list structure.
   * Useful when you need predictable iteration order.
3. **TreeMap**:
   * Stores key-value pairs in a **sorted order** based on the natural ordering of the keys or a custom comparator.
   * Does not allow null keys (but allows null values).
   * Provides log(n) time complexity for most operations.
4. **Hashtable**:
   * Synchronized, thread-safe implementation of a hash map.
   * Does not allow null keys or values.
   * Generally slower than HashMap due to synchronization overhead.

These implementations provide different performance characteristics and are used based on the requirements of ordering, thread-safety, and null support.

Q10. What is the difference between HashMap and TreeMap

| **HashMap** | **TreeMap** |
| --- | --- |
| **1.** | It does not provide any order for elements. | It provides orders for elements. |
| **2.** | It’s speed is fast. | It’s speed is slow. |
| **3.** | It allows one key as null and also allows multiple values. | It does not allow key as null but it allows multiple null values. |
| **4.** | It consumes more memory space. | It consumes less memory space. |
| **5.** | It has only basic features. | It has advanced features. |
| **6.** | For comparing keys, equals() is used. | For comparing keys, compare or compareTo() is used. |
| **7.** | It’s complexity is O(1). | It’s complexity is O(log n). |
|  |  |  |

Q11. How do you check if a key exists in a Map in Java?

To check if a key exists in a **Map** in Java, you can use the containsKey() method. This method returns true if the map contains the specified key, otherwise, it returns false.

Here's an example:

import java.util.HashMap;

import java.util.Map;

public class Main {

public static void main(String[] args) {

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

map.put("A", 1);

map.put("B", 2);

// Check if the key "A" exists

if (map.containsKey("A")) {

System.out.println("Key 'A' exists in the map.");

} else {

System.out.println("Key 'A' does not exist in the map.");

}

}

}