Collections Question Bank

Questions

1. What is collection framework and explain the methods defined by the following interfaces
    1. Collection [Done]
    2. List [Done]
    3. SortedList [Done]
    4. Queue [Done]
2. Explain how Collections can be addressed using an Iterator with an example [Done]
3. Explain HashMap with example [Done]
4. Explain the following collection classes by constructing java program [Done]
    1. LinkedList
    2. ArrayList
    3. Set
    4. HashSet
5. What are legacy classes and explain legacy classes with a java program [Done]
6. Vector legacy class [Done]
7. Explain Constructor of TreeSet with examples [Done]
8. Employee Program
9. List Iterator [Done]
10. Constructors in TreeSet & write a java prog to create a tree set and access it via an iterator [Done]
11. HashMap & Tree Map [Done]

What is collection framework and explain the methods defined by the following interfaces

- Collection Framework gives the programmer the access to prepackaged data structures and algorithms
- A collection is an object that can hold references to otehr objects
- They are stored in java.util package

Memorise the Collections Framework Diagram 3 Marks

1. Collection:
    1. Used to work on groups of objects;
    2. is on top of collections hierarchy
2. DeQueue:
    1. Extends Queue To handle double-ended queues
3. List : Extends Collections to handle Sequences
4. Navigable set: Extends SortedSet to handle navigational features in set
5. Queue: Extends collection to handle Queue DataStructures
6. Set : Extends Collection to handle collection of unique elements
7. SortedSet: Extends Set to handle sorted sets

Collections Interface

Name	Command	Use
add	.add(value)	Adds given value to Collection
clear	.clear()	Removes all elements
contains	.contains(value)	Checks for existance of value in collection
removeAll	.removeAll(collection2)	removes all collection 2 elements from main collection
size	.size()	returns size of collection
toArray	.toArray()	Converts the Collection into an array object
Iterator	`Iterator<E> = .getIterator()`	Returns an Iterator for the collection

List Interface

Name	Commands	Use
get	.get(int index)	returns the element at index
set	.set(int index,E element)	replaces element at specific index
add	.add(int index, E Element)	inserts at specified location
remove	.remove(int index)	Removes element at specified location
index of	.indexOf(Object o)	Returns index of first occurance

SortedSet

Name	Commands	Use
comparator	.comparator()	returns the comparator used to order the elments
first	.first()	returns the first element in the set
last	.last()	returns the last element in the set
headSet	.headSet(E toElement)	returns the set until `toElement`
tailSet	.tailSet(E fromElement)	returns the set start from `fromElement`

Queue

Name	Commands	Use
add	.add(E e)	inserts element into queue
offer	.offer(E e)	inserts and returns true if successful
remove	.remove()	returns and removes head of queue
element	.element()	returns but doesn't remove head of queue
peek	.peek()	returns but doesn't remove head of queue

Explain how Collections can be addressed using an Iterator with an example

An Iterator is an object that enables you to traverse through a collection, one element at a time

Iterator Interface has the following methods

- `hasNext()` : returns true if there are more elements to iterate over
- `next()` : returns the next iterable element in the collection

Code:

```
// Simple Code Snippet ! write full code in exam !
public static void main(String[] args){
    ArrayList<String> Arr = new ArrayList<String>();
    Arr.add("A");
    Arr.add("B");
    Arr.add("C");
    Arr.add("D");

    Iterator<String> itr = Arr.iterator();
    while(itr.hasNext()){
        System.out.println(itr.next());
    }
}
```

```
// Code snippet for listIterator
public static void main(String[] args){
    ArrayList<String> Arr = new ArrayList<String>();
    Arr.add("A");
    Arr.add("B");
    Arr.add("C");
    Arr.add("D");

    Iterator<String> itr = Arr.ListIterator();

    while(itr.hasNext()){
        System.out.println(itr.next());
        System.out.println(itr.previous());
    }
}
```

Explain HashMap with an Example

It provides a way to store key-value pairs, where each key is associated with a specific value.

HashMap uses a hash table to store the map, making it efficient for insertion, deletion, and lookup operations

Constructors:

```
1. HashMap()
2. HashMap(Map<? extends K, ? extends V> m)
3. HashMap(int capacity)
```

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


map.put(1, "One");
map.put(2, "Two");
map.put(3, "Three");


String value = map.get(2);
System.out.println("Value for key 2: " + value);


boolean hasKey = map.containsKey(3);
System.out.println("Does key 3 exist? " + hasKey);


boolean hasValue = map.containsValue("One");
System.out.println("Does value 'One' exist? " + hasValue);

for (Map.Entry<Integer, String> entry : map.entrySet()) {
    System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());
}
```

Explain the following collection classes by constructing java program

Set:

```
import java.util.HashSet;
import java.util.Set;

public class SetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("A");
        set.add("B");
        set.add("C");
        set.add("A");

        System.out.println("Set: " + set);

        set.remove("B");

        System.out.println("Set after removal: " + set);
```

```
        for (String element : set) {
            System.out.println("Element: " + element);
        }
    }
}
```

HashSet:

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> hashSet = new HashSet<>();
        hashSet.add("A");
        hashSet.add("B");
        hashSet.add("C");
        hashSet.add("A");

        System.out.println("HashSet: " + hashSet);

        hashSet.remove("B");

        System.out.println("HashSet after removal: " + hashSet);

        for (String element : hashSet) {
            System.out.println("Element: " + element);
        }
    }
}
```

Legacy Classes

- Vector: Similar to ArrayList but synchronized.
- Stack: A subclass of Vector that implements a last-in, first-out (LIFO) stack
- Hashtable: Similar to HashMap but synchronized and does not allow null keys or values
- Enumeration: An interface for iterating over collections, replaced by Iterator

Vector

```
Vector<String> vector = new Vector<>();
vector.add("A");
vector.add("B");
vector.add("C");
System.out.println("Vector: " + vector);
vector.remove(1);
System.out.println("Vector after removal: " + vector);
```

```
for (String element : vector) {
        System.out.println("Element: " + element);}
```

Stack

```
Stack<String> stack = new Stack<>();
stack.push("A");
stack.push("B");
stack.push("C");
System.out.println("Stack: " + stack);
String topElement = stack.pop();
System.out.println("Popped element: " + topElement);
System.out.println("Stack after pop: " + stack);
for (String element : stack) {
    System.out.println("Element: " + element);}
```

Hashtable

```
Hashtable<Integer, String> hashtable = new Hashtable<>();
hashtable.put(1, "One");
hashtable.put(2, "Two");
hashtable.put(3, "Three");
System.out.println("Hashtable: " + hashtable);
hashtable.remove(2);
System.out.println("Hashtable after removal: " + hashtable);
for (Integer key : hashtable.keySet()) {
    System.out.println("Key: " + key + ", Value: " + hashtable.get(key));
}
```

Enumeration

```
Vector<String> vector = new Vector<>();
vector.add("A");
vector.add("B");
vector.add("C");
Enumeration<String> enumeration = vector.elements();
while (enumeration.hasMoreElements()) {
    String element = enumeration.nextElement();
    System.out.println("Element: " + element);
}
```

Explain Constructors of TreeSet

TreeSet in Java

`TreeSet` is a part of the Java Collections Framework and implements the `NavigableSet` interface, which is a sub-interface of `SortedSet`.

Constructors of TreeSet

1. **TreeSet()**: Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
2. **TreeSet(Collection<? extends E> c)**: Constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.
3. **TreeSet(SortedSet<E> s)**: Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.
4. **TreeSet(Comparator<? super E> comparator)**: Constructs a new, empty tree set, sorted according to the specified comparator.

Example Program to Create a TreeSet and Access it via an Iterator

```
import java.util.Iterator;
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        // Creating a TreeSet with natural ordering
        TreeSet<String> treeSet = new TreeSet<>();

        // Adding elements to the TreeSet
        treeSet.add("Banana");
        treeSet.add("Apple");
        treeSet.add("Mango");
        treeSet.add("Orange");

        // Accessing the TreeSet using an iterator
        Iterator<String> iterator = treeSet.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.println("Element: " + element);
        }
    }
}
```

HashMap & TreeMap

HashMap and TreeMap in Java

HashMap

- **Order**: Does not maintain any order of keys or values.
- **Implementation**: Based on a hash table.
- **Performance**: Provides constant-time performance (O(1)) for basic operations (get, put, remove), assuming a good hash function and no collisions.
- **Null Values**: Allows one null key and multiple null values.

- **Synchronization**: Not synchronized (thread-safe versions can be created externally).

Example of Using HashMap

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> hashMap = new HashMap<>();
        hashMap.put(1, "One");
        hashMap.put(2, "Two");
        hashMap.put(3, "Three");

        for (Map.Entry<Integer, String> entry : hashMap.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }
    }
}
```

TreeMap

- **Order**: Maintains keys in a sorted order (natural ordering or provided by a comparator).
- **Implementation**: Based on a Red-Black tree.
- **Performance**: Provides log-time performance (O(log n)) for basic operations (get, put, remove).
- **Null Values**: Does not allow null keys (throws `NullPointerException`) but allows multiple null values.
- **Synchronization**: Not synchronized (thread-safe versions can be created externally).

Example of Using TreeMap

```
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<Integer, String> treeMap = new TreeMap<>();
        treeMap.put(3, "Three");
        treeMap.put(1, "One");
        treeMap.put(2, "Two");

        for (Map.Entry<Integer, String> entry : treeMap.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
        }
    }
}
```