Collections in Java

What is Collections in Java?

Java provides a comprehensive collections framework that is designed to handle groups of objects. The Java Collections Framework (JCF) includes interfaces, implementations, and algorithms that provide standard ways to handle collections. Here's an overview of the main components of the Java Collections

Framework:

1. Collection:

- The root interface in the collection hierarchy.
- Extended by more specific interfaces like `List`, `Set`, and `Queue`.

2. List:

- An ordered collection (also known as a sequence).
- Allows duplicate elements.
- Examples: `ArrayList`, `LinkedList`, `Vector`.

3. Set:

- A collection that does not allow duplicate elements.
- Examples: `HashSet`, `LinkedHashSet`, `TreeSet`.

4. Queue:

- A collection used to hold multiple elements prior to processing.

- Typically orders elements in FIFO (first-in-first-out) order.
- Examples: `LinkedList`, `PriorityQueue`.

5. Deque:

- A double-ended queue that allows elements to be added or removed from both ends.
- Examples: `ArrayDeque`, `LinkedList`.

6. Map:

- An object that maps keys to values.
- Cannot contain duplicate keys.
- Examples: `HashMap`, `LinkedHashMap`, `TreeMap`, `Hashtable`.

Important Classes and Interfaces:

1. ArrayList:

- A resizable array implementation of the `List` interface.
- Allows random access of elements.

2. LinkedList:

- A doubly linked list implementation of the `List` and `Deque` interfaces.
- Allows sequential access of elements.

3. HashSet:

- A hash table-based implementation of the `Set` interface.
- Does not maintain any order of elements.

4. LinkedHashSet:

- A hash table and linked list implementation of the `Set` interface.
- Maintains insertion order of elements.

5. TreeSet:

- A NavigableSet implementation based on a TreeMap.
- Orders elements using their natural ordering or a specified comparator.

6. HashMap:

- A hash table-based implementation of the `Map` interface.
- Does not maintain any order of keys.

7. LinkedHashMap:

- A hash table and linked list implementation of the `Map` interface.
- Maintains insertion order of keys.

8. TreeMap:

- A NavigableMap implementation based on a Red-Black tree.

- Orders keys using their natural ordering or a specified comparator.

Example:

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        // List example
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Orange");
        System.out.println("List: " + list);

        // Set example
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Orange");
        set.add("Apple"); // Duplicate element
        System.out.println("Set: " + set);

        // Queue example
```

```
        Queue<String> queue = new LinkedList<>();
        queue.add("Apple");
        queue.add("Banana");
        queue.add("Orange");
        System.out.println("Queue: " + queue);
        System.out.println("Poll from queue: " + queue.poll());
        System.out.println("Queue after poll: " + queue);

        // Map example
        Map<String, Integer> map = new HashMap<>();
        map.put("Apple", 1);
        map.put("Banana", 2);
        map.put("Orange", 3);
        System.out.println("Map: " + map);
    }
}
```

1. List Example:

- `ArrayList` is used to create a list of strings.
- `add()` method adds elements to the list.
- `System.out.println() prints the list.

2. Set Example:

- `HashSet` is used to create a set of strings.

- Duplicates are automatically handled (ignored in this case).

- `System.out.println() prints the set.

3. Queue Example:

- `LinkedList` is used to create a queue.

- `add()` method adds elements to the queue.

- `poll()` method removes the head of the queue.

- `System.out.println() prints the queue and the polled element.

4. Map Example:

- `HashMap` is used to create a map that maps strings to integers.

- `put()` method adds key-value pairs to the map.

- `System.out.println() prints the map.

Note: Collections Utility Class

The `Collections` class provides static methods for operating on collections, such as sorting and searching.

import java.util.*;

```
public class CollectionsExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Banana");
        list.add("Apple");
        list.add("Orange");

        Collections.sort(list);
        System.out.println("Sorted List: " + list);

        int index = Collections.binarySearch(list, "Apple");
        System.out.println("Index of 'Apple': " + index);
    }
}
```

1. Collections.sort():

- Sorts the list in natural order.

2. Collections.binarySearch():

- Searches for a specified element in a sorted list.

- Returns the index of the element.

Types of Collections:

- Set
- List
- Map
- Queue
- Stack

SET:

- HashSet
- TreeSet
- LinkedHashSet

HashSet:

- It is a part of Java Collections of Framework and implements the **'Set'** interface.
- No Duplicate elements can be added, only unique elements are added.
- The elements in a '**HashSet**' are not ordered.
- It provides constant time performance for the operations such as add, remove, contains, size.
- It allows single **Null** element.
- We can sort the **HashSet** by converting it into **List** or **TreeSet**.

Operations:

- **add():** Adds the specified element to this set if it is not already present.
- **clear():** Removes all of the elements from this set.
- **contains():** Returns true if this set contains the specified element.
- **isEmpty():** Returns true if this set contains no elements.
- **remove():** Removes the specified element from this set if it is present.
- **int size():** Returns the number of elements in this set.
- **addAll():** Adds all of the elements in the specified collection to this set if they're not already present.
- **containsAll():** Returns true if this set contains all of the elements of the specified collection.
- **removeAll():** Removes from this set all of its elements that are contained in the specified collection.
- **retainAll():** Retains only the elements in this set that are contained in the specified collection.

Example:

```
import java.util.*;
public class Sett {
   public static void main(String[] args) {
      Set <Integer> set = new HashSet<>();
      set.add(3);
      set.add(4);
      set.add(5);
      set.add(0);
      set.add(-3);
      set.add(2);
      set.add(19);
      System.out.println(set);
      System.out.println(set.contains(0));
      System.out.println(set.isEmpty());
      set.remove(0);
      System.out.println(set);
      System.out.println(set.size());
      System.out.println(set);
      set.clear();
```

```
        System.out.println(set);
    }
}
```

Output:

[0, -3, 2, 3, 19, 4, 5]

true

false

[-3, 2, 3, 19, 4, 5]

6

false

[-3, 2, 3, 19, 4, 5]

[ ]

TreeSet:

- TreeSet maintains its elements in sorted order.
- It is Asynchronized.
- It is a part of Java Collections of Framework and implements the **'Set'** interface.
- No Duplicate elements can be added, only unique elements are added.
- It provides constant time performance for the operations such as add, remove, contains, size.

Example:

```
import java.util.*;
public class Sett {
    public static void main(String[] args) {
        TreeSet<String> treeSet = new TreeSet<>();
        treeSet.add("Ariya");
        treeSet.add("Bob");
        treeSet.add("Chandru");

        System.out.println("TreeSet: " + treeSet);
```

```
        treeSet.remove("Bob");

        System.out.println("Contains 'Bob': " + treeSet.contains("Bob"));

        System.out.println("Size of TreeSet: " + treeSet.size());

        System.out.println("Iterating over TreeSet:");
        for (String element : treeSet) {
            System.out.println(element);}}}
```

Output:

TreeSet: [Ariya, Bob, Chandru]

Contains 'Bob': false

Size of TreeSet: 2

Iterating over TreeSet:

Ariya

Chandru

LinkedHashSet:

- In TreeSet order is maintained.
- No Duplicate elements can be added, only unique elements are added.
- It is Asynchronized.
- It provides constant time performance for the operations such as add, remove, contains, size.

Example:

```
        import java.util.*;
public class Sett {
  public static void main(String[] args) {
    Set<String> linkedHashSet = new LinkedHashSet<>();
    linkedHashSet.add("AJSTYLES");
```

```
        linkedHashSet.add("SETH");
        linkedHashSet.add("ROMAN");
        System.out.println("LinkedHashSet: " + linkedHashSet);
        linkedHashSet.remove("SETH");
        System.out.println("Contains 'Banana': " + linkedHashSet.contains("SETH"));
        System.out.println("Size of LinkedHashSet: " + linkedHashSet.size());
        System.out.println("Iterating over LinkedHashSet:");
        for (String element : linkedHashSet) {
            System.out.println(element);
        }
    }
}
```

Output:

LinkedHashSet: [AJSTYLES, SETH, ROMAN]

Contains 'Banana': false

Size of LinkedHashSet: 3

Iterating over LinkedHashSet:

AJSTYLES

SETH

ROMAN

LIST:

- ArrayList
- LinkedList

ArrayList :

- **ArrayList** in Java is a dynamic array that implements the **List** interface. It provides resizable arrays, which can grow or shrink as needed.
- **ArrayList** allows duplicate elements and also permits storing **null** values.
- **ArrayList** maintains the order of insertion of elements. When you iterate over an ArrayList, elements are returned in the order in which they were added.
- **ArrayList** allows fast indexed access and retrieval of elements. Elements can be accessed directly by their index.

Operations:

- **add(E element):** Adds the specified element to the end of the list.
- **add(int index, E element):** Inserts the specified element at the specified position in the list, shifting the subsequent elements to the right.
- **remove(int index):** Removes the element at the specified position in the list.
- **get(int index):** Returns the element at the specified position in the list**.**
- **set(int index, E element):** Replaces the element at the specified position in the list with the specified element.
- **size():** Returns the number of elements in the list.
- **isEmpty():** Returns true if the list contains no elements.
- **clear():** Removes all of the elements from the list.
- **contains(Object o):** Returns true if the list contains the specified element.

- **indexOf(Object o):** Returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
- **lastIndexOf(Object o):** Returns the index of the last occurrence of the specified element in the list, or -1 if the list does not contain the element.
- **toArray():** Returns an array containing all of the elements in the list.

Example:

```
import java.util.*;
public class arrayList {
   public static void main(String[] args) {
      List<Integer>arr=new ArrayList<>();
      arr.add(1);
      arr.add(3);
      arr.add(0);
      arr.add(100);
      arr.add(64);
      System.out.println(arr);
      Collections.sort(arr);
      System.out.println(arr);
      arr.remove(3);
      System.out.println(arr);
      System.out.println(arr.contains(0));
      arr.set(3,50);
      System.out.println(arr.size());
      System.out.println(arr.get(2));
```

```
    }
}
```

Output:

[1, 3, 0, 100, 64]

[0, 1, 3, 64, 100]

[0, 1, 3, 100]

true

4

3

Linked List:

- **Linked List** is a part of the Java Collections Framework.
- Each element in the Doubly LinkedList has references to the next and previous elements, allowing for bi-directional traversal.
- Insertions and deletions of elements are efficient, especially at the beginning and end of the list, as they require only updating the node references. This operation is generally O(1).

- While LinkedList allows indexed access to elements, it is less efficient than ArrayList because it requires traversing the list from the beginning or end to reach the specified index. Access time is O(n) in the worst case.

Example:

```
import java.util.*;
public class LL {
   public static void main(String[] args) {
        LinkedList<String> linkedList = new LinkedList<>();
        linkedList.add("Arian");
        linkedList.add("Berlin");
        linkedList.add("Cal");
        linkedList.addFirst("Men");
        linkedList.addLast("Ooron");
        System.out.println("LinkedList: " + linkedList);
        linkedList.remove("Berlin");
        System.out.println("Contains 'Berlin': " + linkedList.contains("Berlin"));
        System.out.println("Size of LinkedList: " + linkedList.size());
        System.out.println("Iterating over LinkedList:");
        for (String element : linkedList) {
           System.out.println(element);
        }
        System.out.println("First Element: " + linkedList.getFirst());
        System.out.println("Last Element: " + linkedList.getLast());
      }
    }
```

Output:

LinkedList: [Men, Arian, Berlin, Cal, Ooron]

Contains 'Berlin': false

Size of LinkedList: 4

Iterating over LinkedList:

Men

Arian

Cal

Ooron

First Element: Men

Last Element: Ooron

MAP:

- HashMap
- TreeMap

HashMap:

- **HashMap** stores elements in key-value pairs, where each key is unique. A key maps to exactly one value.
- The elements in a **HashMap** are not ordered. The order in which keys and values are stored is not guaranteed to be the same as the order in which they were added.
- **HashMap** provides constant-time performance for basic operations like **get, put** and **remove** assuming the hash function disperses the elements properly across the buckets.
- **HashMap** permits one null key and multiple null values.

Operations:

- **put(K key, V value):** Associates the specified value with the specified key in the map.
- **get(Object key):** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- **remove(Object key):** Removes the mapping for the specified key from this map if present.
- **containsKey(Object key):** Returns true if this map contains a mapping for the specified key.
- **containsValue(Object value):** Returns true if this map maps one or more keys to the specified value.
- **size():** Returns the number of key-value mappings in this map.
- **isEmpty():** Returns true if this map contains no key-value mappings.
- **clear():** Removes all of the mappings from this map.
- **keySet():** Returns a Set view of the keys contained in this map.
- **values():** Returns a Collection view of the values contained in this map.
- **entrySet():** Returns a Set view of the mappings contained in this map.

Example:

```
import java.util.*;
public class hashMap {
  public static void main(String[] args) {
      HashMap<Character,String>map=new HashMap<>();
      map.put('z',"Randy");
      map.put('a',"RomanReigns");
      map.put('c',"BrockLeshnar");
```

```
        map.put('g',"Goldberg");
        map.put('r',"Randy");
        map.put('m',"MIZ");
        map.put('c',"CENA");
        map.remove('r');
    System.out.println(map.get('a'));
    System.out.println(map.containsKey('c'));
    System.out.println(map.size());
        System.out.println(map);
  }
}
```

Output:

RomanReigns

true

5

{a=RomanReigns, c=CENA, g=Goldberg, z=Randy, m=MIZ}.

TreeMap:

- **TreeSet** is a part of the Java Collections Framework.
- It is a map implementation that keeps its entries sorted according to the natural ordering of its keys or according to a specified comparator
- Internally,**TreeMap** is implemented as a Red-Black tree, which guarantees O(log n) time complexity for **get,put and remove** and other basic operations.

- **TreeMap** does not allow null keys (throws **NullPointer Exception**). However, it permits multiple null values.

Example:

```
import java.util.*;
public class LL {
  public static void main(String[] args) {
    TreeMap<Integer, String> treeMap = new TreeMap<>();
    treeMap.put(3, "Undertaker");
    treeMap.put(1, "Roman Reigns");
    treeMap.put(2, "Brock Lesnar");
    treeMap.put(4, "Goldberg");
    treeMap.put(0, "Randy Orton");
    System.out.println("TreeMap: " + treeMap);
    System.out.println("First key: " + treeMap.firstKey());
    System.out.println("Last key: " + treeMap.lastKey());
    System.out.println("Value for key 2: " + treeMap.get(2));
    System.out.println("Higher key than 2: " + treeMap.higherKey(2));
    System.out.println("Lower key than 2: " + treeMap.lowerKey(2));
    Map<Integer, String> subMap = treeMap.subMap(1, 4);
    System.out.println("SubMap from key 1 to key 4: " + subMap);
      }
    }
```

Output:

TreeMap: {0=Randy Orton, 1=Roman Reigns, 2=Brock Lesnar, 3=Undertaker, 4=Goldberg}

First key: 0

Last key: 4

Value for key 2: Brock Lesnar

Higher key than 2: 3

Lower key than 2: 1

SubMap from key 1 to key 4: {1=Roman Reigns, 2=Brock Lesnar, 3=Undertaker}

QUEUE:

- Queue is one of the Collections Framework.
- Elements are processed in the order they are added. The first element added is the first one to be removed.
- It is used in Breadth First Search Algorithm.
- FIFO – First In First Out.

Operations:

- **offer(E e):** Inserts the specified element into the queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and false if no space is currently available.
- **add(E e):** Inserts the specified element into the queue. If the element cannot be added due to capacity restrictions, an IllegalStateException is thrown.
- **poll():** Retrieves and removes the head of the queue, or returns null if the queue is empty.
- **remove():** Retrieves and removes the head of the queue. Throws an exception if the queue is empty.
- **peek():** Retrieves, but does not remove, the head of the queue, or returns null if the queue is empty.
- **element():** Retrieves, but does not remove, the head of the queue. Throws an exception if the queue is empty.

Examples:

```
import java.util.*;
public class queue {
   public static void main(String[] args) {
      Queue<Integer>qu=new LinkedList<>();
     //  System.out.println(qu.remove());  //  NoSuchElementException
      System.out.println(qu.poll());  //shows null if no element
      qu.offer(2);
      qu.add(3);
      qu.offer(43);
      qu.add(37);  // size is full means give exception
      qu.offer(192);  // shows false
      System.out.println(qu);
      System.out.println(qu.peek());  // shows null if no element
      System.out.println(qu.element());  // shows NoSuchElementException
   }
}
```

Output:

null

[2, 3, 43, 37, 192]

2

2

STACK

- **Stack** is a part of the Java Collections Framework.

- Elements are processed in a Last-In-First-Out manner. The last element added (pushed) to the stack is the first one to be removed (popped)
- It is used in Depth First Search Algorithm.
- FILO(First In Last Out) OR LIFO(Last In First Out).

Operations:

- **push(E item):** Pushes an item onto the top of the stack.
- **pop():** Removes the object at the top of the stack and returns that object.
- **peek():** Looks at the object at the top of the stack without removing it from the stack.
- **isEmpty():** Tests if the stack is empty.
- **search(Object o):** Returns the 1-based position where an object is on the stack. If the object is not on the stack, it returns -1.

Example:

```
import java.util.Stack;

public class stack {
  public static void main(String[] args) {
    Stack <Integer> st=new Stack<>();
    st.push(2);
    st.push(5);
    st.push(78);
    st.push(3);
```

```
        st.push(29);
        System.out.println(st.pop());
        st.add(2);
        System.out.println(st.peek());
        System.out.println(st.empty());
        System.out.println(st);
    }
}
```

Output:

29

2

false

[2, 5, 78, 3, 2]

The Java Collections Framework provides a robust and flexible foundation for managing groups of objects, offering numerous functionalities to handle various data structures efficiently.