Collection

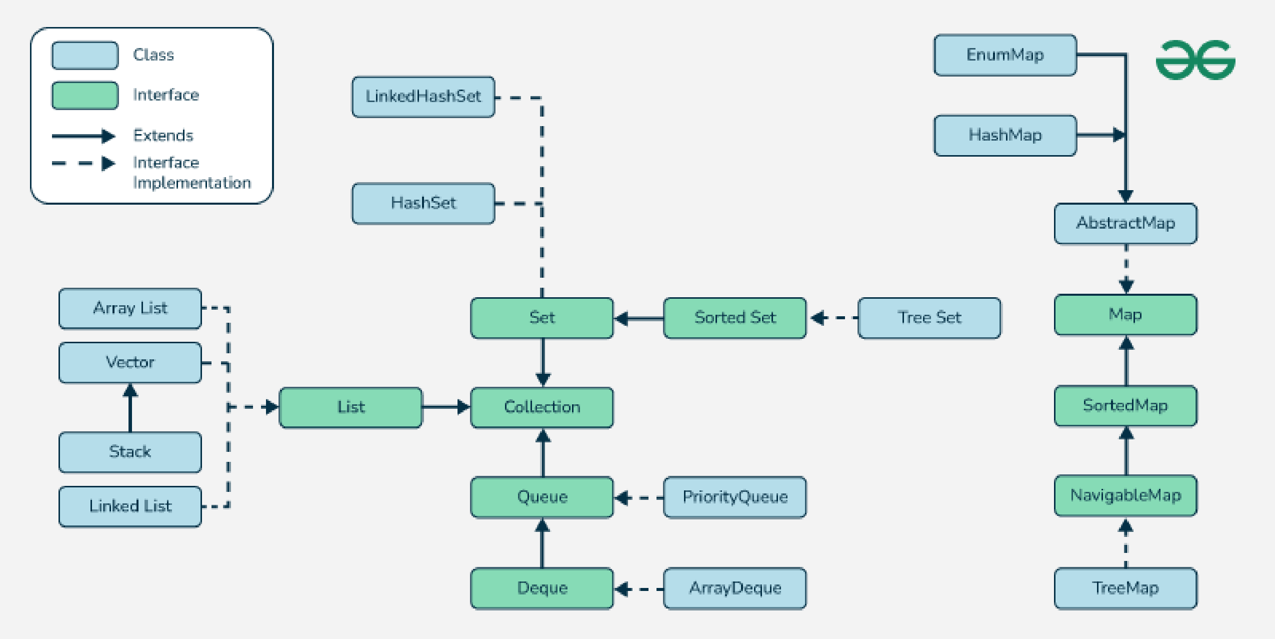
Its a framework using which we can manipulate objects.
Java.util.*;

Why Collections?

- They are dynamic-which means they can grow/shrink in size
- Built-in algo for sorting ,searching etc..

Collection Framework Hierarchy

set of classes and interfaces that implement various types of collections.



Methods Commonly used:

1.add(Object)

2.addAll(Collection c)

3.clear()

4.equals(Object o)

5.hashCode()

6.isEmpty()

7.size()

Interfaces that Extend the Java Collections Interface

Introduction to List Interface

- The **List Interface** is a child interface of the **Collection** interface in Java.
- **Key Characteristics:**
  - **Allows duplicate elements.**
  - Maintains the **insertion order.**

Several classes implement the **List** interface in Java:

- **ArrayList**
- **LinkedList**
- **Vector**
- **Stack**

1)ArrayList

- **ArrayList** is a dynamic array that allows **random access** to elements.
- Resizable, not synchronized, cannot store primitive types

eg:

```
ArrayList<Integer> al = new ArrayList<>();

al.add(1);

al.add(2);

System.out.println(al);  // Output: [1, 2]

al.remove(1);//removes 1

System.out.println(al);  // Output: [1]
```

2)LinkedList

- Uses **nodes**: Each element is a node containing data and a reference to the next node.
- **Efficient for insertions and deletions** at both ends.

Eg:

```
LinkedList<Integer> ll = new LinkedList<>();

ll.add(1);

ll.add(2);

System.out.println(ll);  // Output: [1, 2]

ll.remove(1);

System.out.println(ll);  // Output: [1]
```

3)Vector:

Vector is similar to **ArrayList** but with **synchronization**.

Characteristics:

- **Thread-safe**: Synchronization is provided, making it slower than ArrayList.
- Dynamically resizable, like ArrayList.

Eg:

```
Vector<Integer> v = new Vector<>();

v.add(1);

v.add(2);

System.out.println(v);  // Output: [1, 2]

v.remove(1);

System.out.println(v);  // Output: [1]
```

4)Stack

- **Stack** is a subclass of **Vector** and models the **LIFO (Last-In-First-Out)** data structure.

Operation:
push- Adds an element to the top of the stack.
Pop- Removes the top element.
Peek- Views the top element without removing it.

Eg:

```
Stack<String> stack = new Stack<>();

stack.push("A");

stack.push("B");

stack.pop();  // Removes "B"

System.out.println(stack);  // Output: [A]
```

Introduction to Queue Interface

Queue Interface follows the **FIFO (First-In-First-Out)** principle.
Several classes implement the **Queue** interface in Java:

- **PriorityQueue -** A queue where elements are processed based on **priority**, rather than the order they were added.
- **ArrayDeque - Double-Ended Queue**.- allows elements to be added or removed from both ends.

- **Syntax for creating both:**
  ```
  Queue<Integer> pq = new PriorityQueue<>();
  Queue<Integer> ad = new ArrayDeque<>();
  ```

Introduction to Set Interface

- A **Set** is a collection that **does not allow duplicate values**.
- **Key Characteristics**:

    - Unordered collection of elements.

Set Interface has several classes that implement it:

- **HashSet-** Implements the **hash table** data structure.
- **LinkedHashSet-** Similar to **HashSet**, but it maintains the **insertion order**.
- **TreeSet-** Implements the **SortedSet** interface and stores elements in **sorted order**.

Syntax:

```
Set<T> hs = new HashSet<>();     // HashSet

Set<T> lhs = new LinkedHashSet<>(); // LinkedHashSet

Set<T> ts = new TreeSet<>();      // TreeSet
```

Introduction to Map Interface

- A **Map** is a data structure that stores data in **key-value pairs**.

  No Duplicate Keys
  Duplicate Values Allowed
  Access via Key

The **Map Interface** has several implementing classes:

- **HashMap-** fast and unordered,
- **TreeMap-** sorted order but is slower than **HashMap**.

- **Syntax:**

```
Map<T, V> hm = new HashMap<>();    // HashMap

Map<T, V> tm = new TreeMap<>();    // TreeMap
```

Key Operations in HashMap

1. **put(K key, V value)** – Adds a key-value pair.
2. **get(K key)** – Retrieves the value for a given key.

3. **containsKey(K key)** – Checks if a key is present.
4. **remove(K key)** – Removes the key-value pair for the given key.
5. **size()** – Returns the number of key-value pairs in the map.
6. **entrySet()** – Returns a set of key-value pairs.