Java Collections Framework

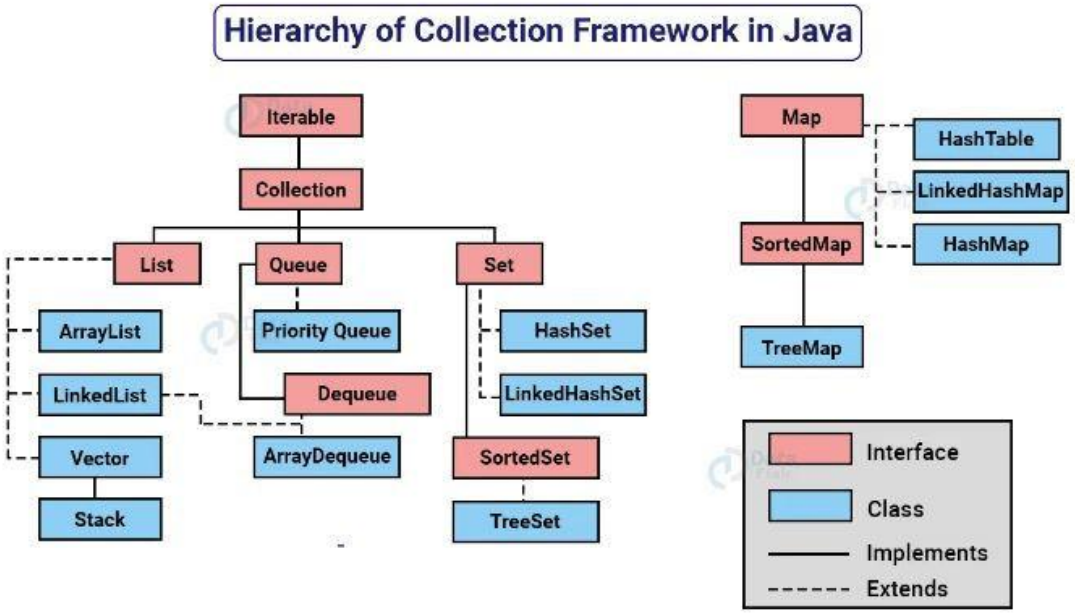
By Fernando Sánchez González

In Java, a **Collection** is a framework that provides an architecture to store and manipulate a group of objects. Collections allow you to perform operations such as searching, sorting, insertion, deletion, and manipulating elements. The Collection framework includes interfaces, implementations (classes), and algorithms to work with different types of data structures.

Key Points:

1. **Interfaces:** The core interfaces in the Collection framework include:
    - **Collection:** The root interface that represents a group of objects.
    - **List:** An ordered collection (like an array) that allows duplicate elements. Examples include ArrayList, LinkedList.
    - **Set:** A collection that does not allow duplicate elements. Examples include HashSet, TreeSet.
    - **Queue:** A collection that orders elements in a FIFO (First-In-First-Out) manner. Examples include LinkedList, PriorityQueue.
    - **Map:** An interface for key-value pairs, though technically not a part of the Collection interface hierarchy. Examples include HashMap, TreeMap.
2. **Classes:** The Collection framework includes various concrete classes that implement these interfaces, such as ArrayList, HashSet, LinkedList, and HashMap.
3. **Algorithms:** The framework provides algorithms to manipulate collections, such as sorting and searching, which are defined as static methods in the Collections utility class.
4. **Generics:** Collections are often used with Java Generics to ensure type safety, meaning you can specify the type of elements stored in a collection, reducing runtime errors.

The Collection framework is a powerful tool in Java, providing a standardized way to handle groups of objects, making data management and manipulation easier and more efficient.



Common Methods in the Collection Interface:

1. **add(E element)**
   Adds the specified element to the collection. Returns true if the collection changed as a result of the operation.
2. **addAll(Collection<? extends E> c)**
   Adds all elements from the specified collection to the current collection. Returns true if the collection changed as a result of the operation.
3. **remove(Object o)**
   Removes a single instance of the specified element from the collection, if it exists. Returns true if the collection contained the element.
4. **removeAll(Collection<?> c)**
   Removes all elements in the current collection that are also contained in the specified collection. Returns true if the collection changed as a result of the operation.
5. **clear()**
   Removes all elements from the collection, leaving it empty.
6. **size()**
   Returns the number of elements in the collection.
7. **isEmpty()**
   Returns true if the collection contains no elements.
8. **contains(Object o)**
   Returns true if the collection contains the specified element.
9. **containsAll(Collection<?> c)**
   Returns true if the collection contains all elements of the specified collection.

10. **iterator()**
    Returns an iterator over the elements in the collection, allowing for traversal of the collection.
11. **toArray()**
    Converts the collection into an array. There are two versions:
    - Object[] toArray(): Returns an array containing all elements in the collection.
    - <T> T[] toArray(T[] a): Returns an array containing all elements in the collection; the runtime type of the returned array is that of the specified array.
12. **retainAll(Collection<?> c)**
    Retains only the elements in the collection that are contained in the specified collection. All others are removed.

ArrayList Class

Key Points of ArrayList:

1. **Dynamic Array:** ArrayList is a resizable array implementation of the List interface, allowing elements to be added or removed dynamically.
2. **Indexed Access:** Elements can be accessed directly by their index, making it efficient for retrieval operations.
3. **Ordered Collection:** Maintains the insertion order of elements.
4. **Allows Duplicates:** ArrayList can contain duplicate elements.
5. **Not Synchronized:** ArrayList is not thread-safe by default; synchronization is needed for concurrent access.

Specific Methods of ArrayList:

1. **add(E element)**
   Adds the specified element to the end of the list.
2. **add(int index, E element)**
   Inserts the specified element at the specified position in the list.
3. **get(int index)**
   Returns the element at the specified position in the list.
4. **set(int index, E element)**
   Replaces the element at the specified position with the specified element.
5. **remove(int index)**
   Removes the element at the specified position in the list. Shifts any subsequent elements to the left.
6. **remove(Object o)**
   Removes the first occurrence of the specified element from the list, if it is present.

7. **clear()**
   Removes all elements from the list, leaving it empty.
8. **subList(int fromIndex, int toIndex)**
   Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
9. **ensureCapacity(int minCapacity)**
   Increases the capacity of the ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
10. **trimToSize()**
    Trims the capacity of the ArrayList instance to the current size, reducing memory usage.

```java

import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        // Create an ArrayList
        ArrayList<String> fruits = new ArrayList<>();

        // Add elements to the ArrayList
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        // Print the ArrayList
        System.out.println("Fruits List: " + fruits);

        // Access elements by index
        System.out.println("First fruit: " + fruits.get(0));

        // Modify an element
        fruits.set(1, "Blueberry");
        System.out.println("Updated Fruits List: " + fruits);

        // Remove an element
        fruits.remove("Cherry");
        System.out.println("Fruits List after removal: " + fruits);

        // Get the index of an element
        int index = fruits.indexOf("Blueberry");
        System.out.println("Index of Blueberry: " + index);

        // Check if the ArrayList contains a specific element
        boolean containsApple = fruits.contains("Apple");
        System.out.println("Contains Apple: " + containsApple);

        // Get the size of the ArrayList
        System.out.println("Size of the list: " + fruits.size());

        // Clear all elements
        fruits.clear();
        System.out.println("Fruits List after clearing: " + fruits);
    }
}

```

In Java, the LinkedList class is part of the Java Collections Framework and implements both the List and Deque interfaces. It represents a doubly-linked list data structure, which allows for efficient insertion and removal of elements at both ends. Here are some key points about LinkedList and its types:

Key Points of LinkedList:

1. **Doubly-Linked List:** Each element (node) in a LinkedList contains references to both the previous and next nodes, allowing traversal in both directions.
2. **Implementations:** Implements List (for ordered collections with indexed access) and Deque (for double-ended queue operations).
3. **Allows Nulls:** LinkedList allows null elements.
4. **Non-Synchronized:** Like most collection classes, LinkedList is not synchronized and needs external synchronization if used in concurrent scenarios.

Types of LinkedList Usage:

1. **As a List:**
    - **Usage:** When used as a List, LinkedList supports operations such as insertion, removal, and access by index.
    - **Example Methods:**
        - add(E e): Adds an element to the end of the list.
        - get(int index): Retrieves the element at the specified position.
        - set(int index, E element): Replaces the element at the specified position.
2. **As a Deque:**
    - **Usage:** When used as a Deque (Double-Ended Queue), LinkedList supports operations to add, remove, and inspect elements at both ends of the list.
    - **Example Methods:**
        - addFirst(E e): Inserts the specified element at the beginning.
        - addLast(E e): Appends the specified element to the end.
        - removeFirst(): Removes and returns the first element.
        - removeLast(): Removes and returns the last element.
        - peekFirst(): Retrieves, but does not remove, the first element.
        - peekLast(): Retrieves, but does not remove, the last element.

```java
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Create a LinkedList
        LinkedList<String> list = new LinkedList<>();

        // Using LinkedList as a List
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");
        System.out.println("LinkedList as List: " + list);

        // Accessing and modifying elements
        System.out.println("Element at index 1: " + list.get(1));
        list.set(1, "Blueberry");
        System.out.println("Updated LinkedList as List: " + list);

        // Using LinkedList as a Deque
        list.addFirst("Orange");
        list.addLast("Grapes");
        System.out.println("LinkedList as Deque: " + list);

        // Removing elements
        list.removeFirst();
        list.removeLast();
        System.out.println("LinkedList after removing elements: " + list);

        // Peeking elements
        System.out.println("First element: " + list.peekFirst());
        System.out.println("Last element: " + list.peekLast());
    }
```

Queue Interface

The Queue interface in Java is part of the Java Collections Framework and represents a collection designed for holding elements prior to processing. It supports operations for adding, removing, and inspecting elements in a first-in-first-out (FIFO) order, though some implementations may support other ordering schemes.

Key Points of the Queue Interface:

1. **FIFO Ordering:** The primary ordering scheme for most queue implementations is FIFO, meaning elements are processed in the order they were added.
2. **Blocking Operations:** Some queue implementations (like BlockingQueue) support operations that wait for conditions to become true, such as when the queue is empty or full.

3. **Flexible Implementations:** Several classes implement the Queue interface, including LinkedList, PriorityQueue, and ArrayDeque, each with different characteristics and performance trade-offs.
4. **Non-Synchronized:** The Queue interface itself does not guarantee synchronization; thread safety must be managed by the specific implementation or through additional synchronization.

Common Methods of the Queue Interface:

1. **add(E e)**
   Inserts the specified element into the queue. Returns true if the element was added successfully. Throws an exception if the queue is full.
2. **offer(E e)**
   Inserts the specified element into the queue. Returns true if the element was added successfully. Returns false if the queue is full (does not throw an exception).
3. **remove()**
   Removes and returns the element at the front of the queue. Throws an exception if the queue is empty.
4. **poll()**
   Removes and returns the element at the front of the queue. Returns null if the queue is empty.
5. **peek()**
   Retrieves, but does not remove, the element at the front of the queue. Returns null if the queue is empty.
6. **element()**
   Retrieves, but does not remove, the element at the front of the queue. Throws an exception if the queue is empty.
7. **size()**
   Returns the number of elements currently in the queue.
8. **isEmpty()**
   Returns true if the queue contains no elements.

PriorityQueue Class

Key Points:

- **Priority-Based Ordering:** Elements are ordered based on their natural ordering or by a Comparator provided at queue construction.
- **No Capacity Limit:** Unlike some other queues, PriorityQueue does not have a fixed capacity and can grow as needed.
- **Not Thread-Safe:** PriorityQueue is not synchronized and should be used in a thread-safe manner if accessed by multiple threads.
- **Does Not Allow Nulls:** Null elements are not allowed and will throw NullPointerException if added.

- **Queue Interface:** Implements the Queue interface and is a priority-based collection rather than a FIFO queue.

 Common Methods:

- **add(E e)** - Inserts the specified element into the queue. Returns true if the element was added successfully.
- **offer(E e)** - Inserts the specified element into the queue. Returns true if the element was added successfully; false if the queue cannot accept more elements.
- **remove()** - Removes and returns the element with the highest priority (i.e., the lowest element according to the natural ordering or the comparator). Throws an exception if the queue is empty.
- **poll()** - Removes and returns the element with the highest priority. Returns null if the queue is empty.
- **peek()** - Retrieves, but does not remove, the element with the highest priority. Returns null if the queue is empty.
- **size()** - Returns the number of elements in the queue.
- **isEmpty()** - Returns true if the queue contains no elements.

```java
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        // Create a PriorityQueue
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Add elements to the PriorityQueue
        pq.add(50);
        pq.add(20);
        pq.add(30);
        pq.add(10);

        // Print the PriorityQueue
        System.out.println("PriorityQueue: " + pq);

        // Peek at the highest-priority element
        System.out.println("Highest-priority element: " + pq.peek());

        // Remove and print the highest-priority element
        System.out.println("Removed element: " + pq.poll());

        // Print the PriorityQueue after removal
        System.out.println("PriorityQueue after removal: " + pq);

        // Check the size of the PriorityQueue
        System.out.println("Size of the PriorityQueue: " + pq.size());

        // Check if the PriorityQueue is empty
        System.out.println("Is PriorityQueue empty? " + pq.isEmpty());
    }
}
```

Deque Interface

- **Key Points:**
  - **Double-Ended Queue:** Supports insertion and removal of elements from both ends (front and back) of the queue.
  - **FIFO and LIFO:** Can be used as both a queue (FIFO) and a stack (LIFO).
  - **Not Synchronized:** Deque implementations are not thread-safe unless explicitly synchronized.
  - **Flexible Implementations:** Several classes, such as LinkedList and ArrayDeque, implement the Deque interface.

    - **Allows Nulls:** Unlike PriorityQueue, most Deque implementations allow null elements.
- **Common Methods:**
    - **addFirst(E e)** - Inserts the specified element at the front of the deque.
    - **addLast(E e)** - Appends the specified element to the end of the deque.
    - **offerFirst(E e)** - Inserts the specified element at the front of the deque, returns true if successful.
    - **offerLast(E e)** - Appends the specified element to the end of the deque, returns true if successful.
    - **removeFirst()** - Removes and returns the first element of the deque. Throws an exception if the deque is empty.
    - **removeLast()** - Removes and returns the last element of the deque. Throws an exception if the deque is empty.
    - **pollFirst()** - Removes and returns the first element of the deque. Returns null if the deque is empty.
    - **pollLast()** - Removes and returns the last element of the deque. Returns null if the deque is empty.
    - **peekFirst()** - Retrieves, but does not remove, the first element of the deque. Returns null if the deque is empty.
    - **peekLast()** - Retrieves, but does not remove, the last element of the deque. Returns null if the deque is empty.
    - **size()** - Returns the number of elements in the deque.
    - **isEmpty()** - Returns true if the deque contains no elements.

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        // Create a Deque
        Deque<String> deque = new ArrayDeque<>();

        // Add elements to the Deque
        deque.addFirst("First");
        deque.addLast("Last");
        deque.addLast("Middle");

        // Print the Deque
        System.out.println("Deque: " + deque);

        // Peek at the first and last elements
        System.out.println("First element: " + deque.peekFirst());
        System.out.println("Last element: " + deque.peekLast());

        // Remove and print elements from both ends
        System.out.println("Removed first element: " + deque.removeFirst());
        System.out.println("Removed last element: " + deque.removeLast());

        // Print the Deque after removals
        System.out.println("Deque after removals: " + deque);

        // Check the size of the Deque
        System.out.println("Size of the Deque: " + deque.size());

        // Check if the Deque is empty
        System.out.println("Is Deque empty? " + deque.isEmpty());
    }
```

Set Interface

- **Key Points:**
  - **Unique Elements:** A Set is a collection that does not allow duplicate elements. Each element must be unique.
  - **Unordered:** Elements in a Set are not stored in any particular order, though some implementations, like LinkedHashSet, maintain insertion order.
  - **No Indexes:** Unlike List, Set does not provide methods to access elements by index.
  - **Implementations:** Common implementations include HashSet (which offers constant time performance for basic operations), LinkedHashSet (which maintains insertion order), and TreeSet (which sorts elements based on their natural ordering or a comparator).

    - **Not Synchronized:** Set implementations are not synchronized and need to be synchronized externally if used in concurrent scenarios.

Common Methods:

- **add(E e)**
  Inserts the specified element into the set. Returns true if the set did not already contain the element.
- **remove(Object o)**
  Removes the specified element from the set if it is present. Returns true if the set contained the specified element.
- **contains(Object o)**
  Returns true if the set contains the specified element.
- **size()**
  Returns the number of elements in the set.
- **isEmpty()**
  Returns true if the set contains no elements.
- **clear()**
  Removes all elements from the set.
- **iterator()**
  Returns an iterator over the elements in the set.
- **addAll(Collection<? extends E> c)**
  Adds all elements from the specified collection to the set.
- **removeAll(Collection<?> c)**
  Removes from the set all elements that are contained in the specified collection.
- **retainAll(Collection<?> c)**
  Retains only the elements in the set that are contained in the specified collection.
- **containsAll(Collection<?> c)**
  Returns true if the set contains all elements of the specified collection.

HashSet Class

- **Key Points:**
    - **Hashing-Based:** Implements the Set interface using a hash table, which provides constant time performance for basic operations like add, remove, and contains.
    - **No Order:** Does not guarantee any specific order of elements. The order may change over time.
    - **Allows Nulls:** Allows at most one null element.
    - **Not Synchronized:** HashSet is not synchronized and should be synchronized externally if used in a multi-threaded environment.
- **Common Methods:**
    - **add(E e)** - Inserts the specified element into the set if it is not already present.

- **remove(Object o)** - Removes the specified element from the set if it is present.
- **contains(Object o)** - Returns true if the set contains the specified element.
- **size()** - Returns the number of elements in the set.
- **isEmpty()** - Returns true if the set contains no elements.
- **clear()** - Removes all elements from the set.
- **iterator()** - Returns an iterator over the elements in the set.

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

public class Main {
    public static void main(String[] args) {
        // Create a HashSet
        Set<String> set = new HashSet<>();

        // Add elements to the HashSet
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Date");
        set.add("Apple"); // Duplicate element, will not be added

        // Print the HashSet
        System.out.println("HashSet: " + set);

        // Check if the set contains a specific element
        System.out.println("Contains 'Banana'? " + set.contains("Banana"));

        // Remove an element from the HashSet
        set.remove("Date");
        System.out.println("HashSet after removal: " + set);

        // Check the size of the HashSet
        System.out.println("Size of the HashSet: " + set.size());

        // Check if the HashSet is empty
        System.out.println("Is HashSet empty? " + set.isEmpty());

        // Clear all elements from the HashSet
        set.clear();
        // Check the size of the HashSet
        System.out.println("Size of the HashSet: " + set.size());

        // Check if the HashSet is empty
        System.out.println("Is HashSet empty? " + set.isEmpty());

        // Clear all elements from the HashSet
        set.clear();
        System.out.println("HashSet after clearing: " + set);
    }
}
```

LinkedHashSet Class

- **Key Points:**
  - **Hashing with Order:** Implements the Set interface using a hash table with a linked list to maintain insertion order.
  - **Maintains Order:** Preserves the order in which elements are inserted into the set.
  - **Allows Nulls:** Allows at most one null element.
  - **Not Synchronized:** LinkedHashSet is not synchronized and should be synchronized externally if used in a multi-threaded environment.
- **Common Methods:**
  - **add(E e)** - Inserts the specified element into the set if it is not already present.
  - **remove(Object o)** - Removes the specified element from the set if it is present.
  - **contains(Object o)** - Returns true if the set contains the specified element.
  - **size()** - Returns the number of elements in the set.
  - **isEmpty()** - Returns true if the set contains no elements.
  - **clear()** - Removes all elements from the set.
  - **iterator()** - Returns an iterator over the elements in the set, in insertion order.

```java
package Outpout;

import java.util.LinkedHashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        // Create a LinkedHashSet
        Set<String> set = new LinkedHashSet<>();

        // Add elements to the LinkedHashSet
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Date");
        set.add("Apple"); // Duplicate element, will not be added

        // Print the LinkedHashSet
        System.out.println("LinkedHashSet: " + set);

        // Check if the set contains a specific element
        System.out.println("Contains 'Banana'? " + set.contains("Banana"));

        // Remove an element from the LinkedHashSet
        set.remove("Date");
        System.out.println("LinkedHashSet after removal: " + set);

        // Print the LinkedHashSet
        System.out.println("LinkedHashSet: " + set);

        // Check if the set contains a specific element
        System.out.println("Contains 'Banana'? " + set.contains("Banana"));

        // Remove an element from the LinkedHashSet
        set.remove("Date");
        System.out.println("LinkedHashSet after removal: " + set);

        // Check the size of the LinkedHashSet
        System.out.println("Size of the LinkedHashSet: " + set.size());

        // Check if the LinkedHashSet is empty
        System.out.println("Is LinkedHashSet empty? " + set.isEmpty());

        // Clear all elements from the LinkedHashSet
        set.clear();
        System.out.println("LinkedHashSet after clearing: " + set);
    }
}
```

```
<terminated> Main (4) [Java Application] C:\Users\HP\.p2\pool\plugins\org
LinkedHashSet: [Apple, Banana, Cherry, Date]
Contains 'Banana'? true
LinkedHashSet after removal: [Apple, Banana, Cherry]
Size of the LinkedHashSet: 3
Is LinkedHashSet empty? false
LinkedHashSet after clearing: []
```

SortedSet Interface

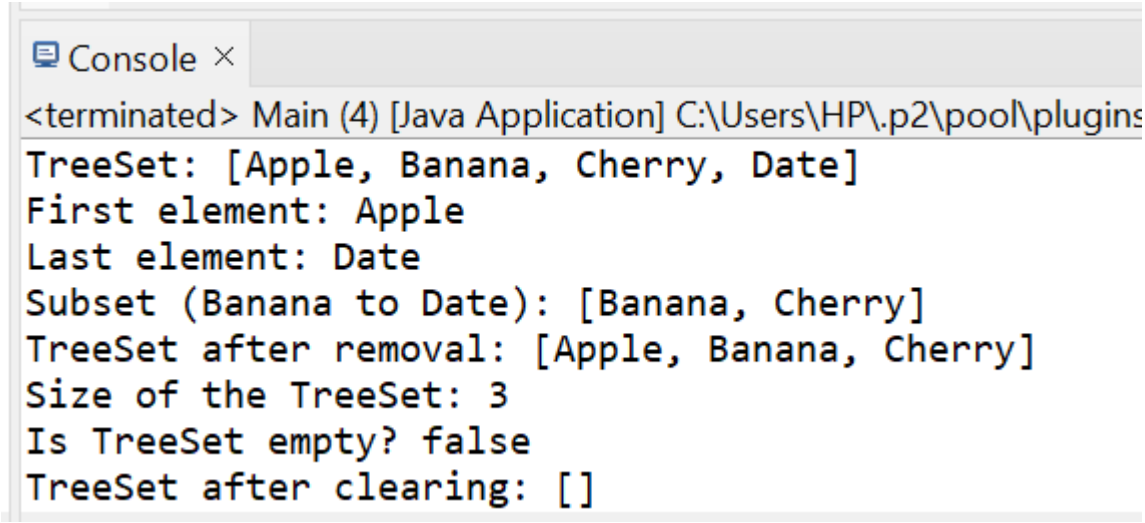
- **Key Points:**
  - **Sorted Order:** Extends the Set interface to handle elements in a sorted order. Elements are maintained in their natural ordering or by a Comparator provided at set creation.
  - **Subset Operations:** Provides methods to view a portion of the set, such as headSet, tailSet, and subSet, based on sorting.
  - **Navigable Operations:** Enables navigation through the set with methods such as first(), last(), lower(), higher(), etc.
  - **No Duplicates:** Does not allow duplicate elements and ensures elements are sorted.
- **Common Methods:**
  - **first()**
    Returns the first (lowest) element in the sorted set.
  - **last()**
    Returns the last (highest) element in the sorted set.
  - **headSet(E toElement)**
    Returns a view of the portion of this set whose elements are strictly less than toElement.
  - **tailSet(E fromElement)**
    Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
  - **subSet(E fromElement, E toElement)**
    Returns a view of the portion of this set between fromElement (inclusive) and toElement (exclusive).
  - **comparator()**
    Returns the comparator used to sort the elements, or null if the set is sorted by natural ordering.
  - **contains(Object o)**
    Returns true if the set contains the specified element.
  - **add(E e)**
    Inserts the specified element into the set if it is not already present.
  - **remove(Object o)**
    Removes the specified element from the set if it is present.

- **size()**
  Returns the number of elements in the set.
- **isEmpty()**
  Returns true if the set contains no elements.
- **clear()**
  Removes all elements from the set.

TreeSet Class

- **Key Points:**
  - **Sorted Set:** Implements the SortedSet interface and uses a Red-Black tree to maintain elements in sorted order.
  - **Natural Ordering or Comparator:** Elements are ordered according to their natural ordering or by a Comparator provided at set creation.
  - **No Duplicates:** Does not allow duplicate elements.
  - **Navigable:** Provides methods for navigating through the set, such as first(), last(), lower(), higher(), etc.
  - **Not Synchronized:** TreeSet is not synchronized and should be synchronized externally if used in a multi-threaded environment.
- **Common Methods:**
  - **add(E e)** - Inserts the specified element into the set if it is not already present.
  - **remove(Object o)** - Removes the specified element from the set if it is present.
  - **contains(Object o)** - Returns true if the set contains the specified element.
  - **first()** - Returns the first (lowest) element in the sorted set.
  - **last()** - Returns the last (highest) element in the sorted set.
  - **headSet(E toElement)** - Returns a view of the portion of this set whose elements are strictly less than toElement.
  - **tailSet(E fromElement)** - Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
  - **subSet(E fromElement, E toElement)** - Returns a view of the portion of this set between fromElement (inclusive) and toElement (exclusive).
  - **comparator()** - Returns the comparator used to sort the elements, or null if the set is sorted by natural ordering.

```java
package Outpout;

import java.util.TreeSet;

public class Main {
    public static void main(String[] args) {
        // Create a TreeSet
        TreeSet<String> treeSet = new TreeSet<>();

        // Add elements to the TreeSet
        treeSet.add("Apple");
        treeSet.add("Banana");
        treeSet.add("Cherry");
        treeSet.add("Date");
        treeSet.add("Apple"); // Duplicate element, will not be added

        // Print the TreeSet
        System.out.println("TreeSet: " + treeSet);

        // Get the first and last elements
        System.out.println("First element: " + treeSet.first());
        System.out.println("Last element: " + treeSet.last());

        // Get a subset of the TreeSet
        System.out.println("TreeSet: " + treeSet);

        // Get the first and last elements
        System.out.println("First element: " + treeSet.first());
        System.out.println("Last element: " + treeSet.last());

        // Get a subset of the TreeSet
        System.out.println("Subset (Banana to Date): " + treeSet.subSet("Banana", "Date"));

        // Remove an element from the TreeSet
        treeSet.remove("Date");
        System.out.println("TreeSet after removal: " + treeSet);

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

        // Check if the TreeSet is empty
        System.out.println("Is TreeSet empty? " + treeSet.isEmpty());

        // Clear all elements from the TreeSet
        treeSet.clear();
        System.out.println("TreeSet after clearing: " + treeSet);
    }
}
```



Map Interface

- **Key Points:**
  - **Key-Value Pairs:** A Map is a collection of key-value pairs, where each key is unique, and each key maps to exactly one value.
  - **No Duplicate Keys:** Keys in a Map must be unique. If a key already exists, adding a new key-value pair with the same key will replace the old value with the new one.
  - **No Index-Based Access:** Unlike lists, Map does not provide access to elements by index but rather by key.
  - **Implementations:** Common implementations include HashMap, LinkedHashMap, and TreeMap, each providing different behaviors for ordering and performance.
  - **Not Synchronized:** Most Map implementations are not synchronized and require external synchronization in multi-threaded contexts.

Common Methods:

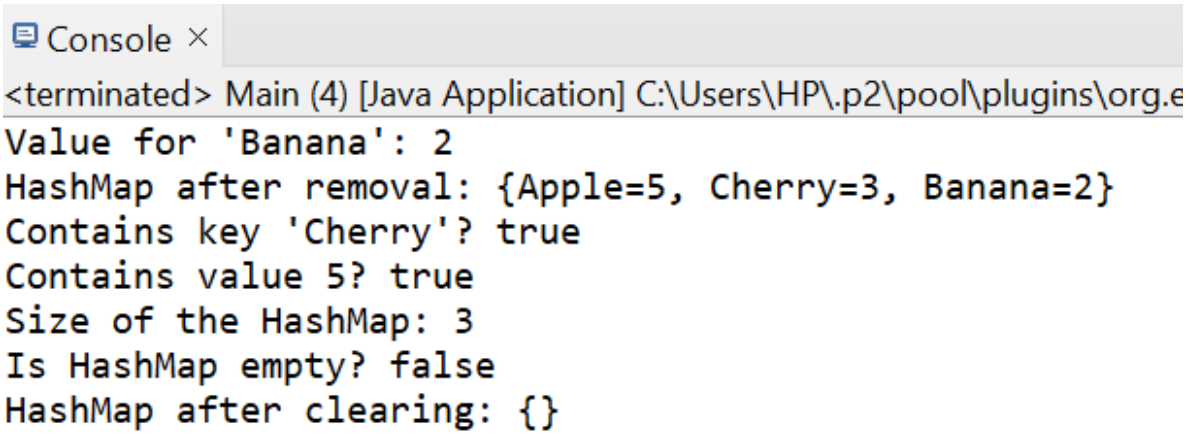
- **put(K key, V value)**
  Inserts the specified key-value pair into the map. If the key already exists, updates the value.
- **get(Object key)**
  Retrieves the value associated with the specified key.
- **remove(Object key)**
  Removes the key-value pair associated with the specified key.
- **containsKey(Object key)**
  Returns true if the map contains the specified key.

- **containsValue(Object value)**
  Returns true if the map contains one or more keys associated with the specified value.
- **size()**
  Returns the number of key-value pairs in the map.
- **isEmpty()**
  Returns true if the map contains no key-value pairs.
- **clear()**
  Removes all key-value pairs from the map.
- **keySet()**
  Returns a Set view of the keys contained in the map.
- **values()**
  Returns a Collection view of the values contained in the map.
- **entrySet()**
  Returns a Set view of the key-value pairs (entries) contained in the map.

HashMap Class

- **Key Points:**
  - **Hashing-Based:** Implements the Map interface using a hash table, which allows for constant time performance for basic operations like add, remove, and contains.
  - **No Order:** Does not guarantee any specific order of its elements. The order may vary and is not predictable.
  - **Allows Nulls:** Allows one null key and multiple null values.
  - **Not Synchronized:** HashMap is not synchronized. It should be synchronized externally if used in a multi-threaded environment.
- **Common Methods:**
  - **put(K key, V value)** - Inserts the specified key-value pair into the map. If the key already exists, updates the value.
  - **get(Object key)** - Retrieves the value associated with the specified key.
  - **remove(Object key)** - Removes the key-value pair associated with the specified key.
  - **containsKey(Object key)** - Returns true if the map contains the specified key.
  - **containsValue(Object value)** - Returns true if the map contains one or more keys associated with the specified value.
  - **size()** - Returns the number of key-value pairs in the map.
  - **isEmpty()** - Returns true if the map contains no key-value pairs.
  - **clear()** - Removes all key-value pairs from the map.
  - **keySet()** - Returns a Set view of the keys contained in the map.
  - **values()** - Returns a Collection view of the values contained in the map.
  - **entrySet()** - Returns a Set view of the key-value pairs (entries) contained in the map.

```java
package Outpout;

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

public class Main {
    public static void main(String[] args) {
        // Create a HashMap
        Map<String, Integer> map = new HashMap<>();

        // Add key-value pairs to the HashMap
        map.put("Apple", 1);
        map.put("Banana", 2);
        map.put("Cherry", 3);
        map.put("Date", 4);
        map.put("Apple", 5); // Duplicate key, will update the value

        // Print the HashMap
        System.out.println("HashMap: " + map);

        // Get a value by key
        System.out.println("Value for 'Banana': " + map.get("Banana"));

        // Remove a key-value pair
        map.remove("Date");
        System.out.println("HashMap: " + map);

        // Get a value by key
        System.out.println("Value for 'Banana': " + map.get("Banana"));

        // Remove a key-value pair
        map.remove("Date");
        System.out.println("HashMap after removal: " + map);

        // Check if the map contains a specific key and value
        System.out.println("Contains key 'Cherry'? " + map.containsKey("Cherry"));
        System.out.println("Contains value 5? " + map.containsValue(5));

        // Get the size of the HashMap
        System.out.println("Size of the HashMap: " + map.size());

        // Check if the HashMap is empty
        System.out.println("Is HashMap empty? " + map.isEmpty());

        // Clear all key-value pairs
        map.clear();
        System.out.println("HashMap after clearing: " + map);
    }
}
```



LinkedHashMap Class

- **Key Points:**
  - **Order of Insertion:** Extends HashMap to maintain insertion order of elements, or access order if specified.
  - **Hashing-Based:** Uses a hash table combined with a linked list to maintain order.
  - **Allows Nulls:** Permits one null key and multiple null values.
  - **Not Synchronized:** Not thread-safe; synchronization is required for concurrent use.
- **Common Methods:**
  - **put(K key, V value)** - Inserts or updates the key-value pair.
  - **get(Object key)** - Retrieves the value associated with the specified key.
  - **remove(Object key)** - Removes the key-value pair by key.
  - **containsKey(Object key)** - Checks if the map contains the specified key.
  - **containsValue(Object value)** - Checks if the map contains the specified value.
  - **size()** - Returns the number of key-value pairs.
  - **isEmpty()** - Returns true if the map is empty.
  - **clear()** - Removes all key-value pairs from the map.
  - **keySet()** - Returns a set of keys.
  - **values()** - Returns a collection of values.
  - **entrySet()** - Returns a set of key-value pairs.

```java
package Outpout;
import java.util.LinkedHashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        // Create a LinkedHashMap
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

        // Add key-value pairs to the LinkedHashMap
        linkedHashMap.put("Apple", 1);
        linkedHashMap.put("Banana", 2);
        linkedHashMap.put("Cherry", 3);
        linkedHashMap.put("Date", 4);
        linkedHashMap.put("Apple", 5); // Duplicate key, updates the value

        // Print the LinkedHashMap
        System.out.println("LinkedHashMap: " + linkedHashMap);

        // Get a value by key
        System.out.println("Value for 'Banana': " + linkedHashMap.get("Banana"));

        // Remove a key-value pair
        linkedHashMap.remove("Date");
        System.out.println("LinkedHashMap after removal: " + linkedHashMap);

        // Check if the map contains a specific key and value
        System.out.println("Contains key 'Cherry'? " + linkedHashMap.containsKey("Cherry"));
        System.out.println("Contains value 5? " + linkedHashMap.containsValue(5));

        // Get the size of the LinkedHashMap
        System.out.println("Size of the LinkedHashMap: " + linkedHashMap.size());

        // Check if the LinkedHashMap is empty
        System.out.println("Is LinkedHashMap empty? " + linkedHashMap.isEmpty());

        // Clear all key-value pairs
        linkedHashMap.clear();
        System.out.println("LinkedHashMap after clearing: " + linkedHashMap);
    }
}
```

Console ×

<terminated> Main (4) [Java Application] C:\Users\HP\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_22.0.2.v2

```
Value for 'Banana': 2
LinkedHashMap after removal: {Apple=5, Banana=2, Cherry=3}
Contains key 'Cherry'? true
Contains value 5? true
Size of the LinkedHashMap: 3
Is LinkedHashMap empty? false
LinkedHashMap after clearing: {}
```

TreeMap Class

- **Key Points:**
  - **Sorted Map:** Implements the NavigableMap interface and sorts its keys according to their natural order or by a Comparator provided at map creation.
  - **Red-Black Tree:** Uses a Red-Black tree to maintain sorted order.
  - **No Null Keys:** Does not allow null keys (throws NullPointerException if attempted), but allows null values.
  - **Navigable Operations:** Provides additional methods to perform navigable operations like finding the closest matches.
- **Common Methods:**
  - **put(K key, V value)** - Inserts or updates the key-value pair.
  - **get(Object key)** - Retrieves the value associated with the specified key.
  - **remove(Object key)** - Removes the key-value pair by key.
  - **containsKey(Object key)** - Checks if the map contains the specified key.
  - **containsValue(Object value)** - Checks if the map contains the specified value.
  - **size()** - Returns the number of key-value pairs.
  - **isEmpty()** - Returns true if the map is empty.
  - **clear()** - Removes all key-value pairs from the map.
  - **firstKey()** - Returns the first (lowest) key.
  - **lastKey()** - Returns the last (highest) key.
  - **higherKey(K key)** - Returns the least key greater than the given key.
  - **lowerKey(K key)** - Returns the greatest key less than the given key.

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

public class Main {
    public static void main(String[] args) {
        // Create a TreeMap
        Map<String, Integer> treeMap = new TreeMap<>();

        // Add key-value pairs to the TreeMap
        treeMap.put("Apple", 1);
        treeMap.put("Banana", 2);
        treeMap.put("Cherry", 3);
        treeMap.put("Date", 4);
        treeMap.put("Fig", 5);

        // Print the TreeMap
        System.out.println("TreeMap: " + treeMap);

        // Get a value by key
        System.out.println("Value for 'Banana': " + treeMap.get("Banana"));

        // Remove a key-value pair
        treeMap.remove("Date");
        System.out.println("TreeMap after removal: " + treeMap);

        // Check if the map contains a specific key and value
        System.out.println("Contains key 'Cherry'? " + treeMap.containsKey("Cherry"));
        System.out.println("Contains value 5? " + treeMap.containsValue(5));

        // Get the size of the TreeMap
        System.out.println("Size of the TreeMap: " + treeMap.size());
```

```
        System.out.println("TreeMap: " + treeMap);

        // Get a value by key
        System.out.println("Value for 'Banana': " + treeMap.get("Banana"));

        // Remove a key-value pair
        treeMap.remove("Date");
        System.out.println("TreeMap after removal: " + treeMap);

        // Check if the map contains a specific key and value
        System.out.println("Contains key 'Cherry'? " + treeMap.containsKey("Cherry"));
        System.out.println("Contains value 5? " + treeMap.containsValue(5));

        // Get the size of the TreeMap
        System.out.println("Size of the TreeMap: " + treeMap.size());

        // Check if the TreeMap is empty
        System.out.println("Is TreeMap empty? " + treeMap.isEmpty());

        // Retrieve the first and last keys
        System.out.println("First key: " + treeMap.firstKey());
        System.out.println("Last key: " + treeMap.lastKey());

        // Retrieve keys higher and lower than a given key
        System.out.println("Key higher than 'Banana': " + treeMap.higherKey("Banana"));
        System.out.println("Key lower than 'Banana': " + treeMap.lowerKey("Banana"));

        // Clear all key-value pairs
        treeMap.clear();
        System.out.println("TreeMap after clearing: " + treeMap);
    }
}
```