Collections in Java

Karen Giselle Valdez Muñoz

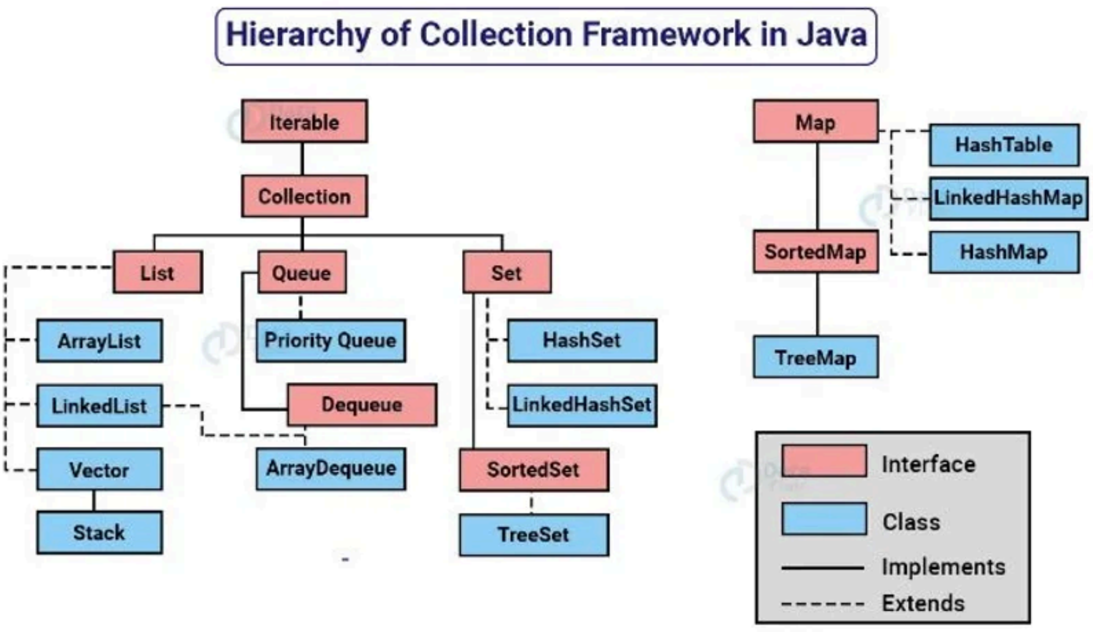
INDEX

Collections

A **Collection** is a framework that provides an architecture to store and manipulate groups of objects. Collections are used to perform operations such as searching, sorting, insertion, manipulation, and deletion.

The Java Collections Framework (JCF) the Collection interface, Map interface and Iterable interface. The Collection interface includes a set of interfaces (Set, List, Queue) each of these interfaces defines the structure and behavior of a collection type, and are implemented by the classes contained in them.

The Collections Framework provides several algorithms that can be applied to collections, such as sorting, searching, and shuffling. These are usually implemented as static methods in the Collections class.



List interface

Key points:
- Maintains the order of its elements. Elements can be accessed by their position (index) in the list, starting from zero.
- Provides methods to access, insert, and remove elements based on their index in the list.
- It can contain duplicate elements. This means that the same element can appear multiple times within the list.
- It can include null elements, depending on the implementation.

- Elements in a List can be iterated using an enhanced for-loop, a standard for loop with an index, or an Iterator.
- It has 2 commonly used classes: ArrayList and LinkedList.

  Common Methods:
    - *boolean add(E e)*: Adds the specified element to the end of the list.
    - *void add(int index, E element)*: Inserts the specified element at the specified position in the list.
    - *boolean remove(Object o)*: Removes the first occurrence of the specified element from the list.
    - *E remove(int index):* Removes the element at the specified position in the list.
    - *E get(int index)*: Returns the element at the specified position in the list.
    - *void set(int index, E element)*: Replaces the element at the specified position in the list with the specified element.
    - *int size():* Returns the number of elements in the list.
    - *void clear()*:This method is used to remove all the elements from any list.
    - *boolean isEmpty()*: Returns true if the list contains no elements.
    - *boolean contains(Object o)*: Returns true if the list contains the specified element.
    - *int indexOf(Object o):* Returns the index of the first occurrence of the specified element, or -1 if the list does not contain the element.
    - *clone()*: This method is used to return a shallow copy of an ArrayList in Java.
    - *forEach(Consumer< super E> action)*: Returns the element at the specified position in this list.

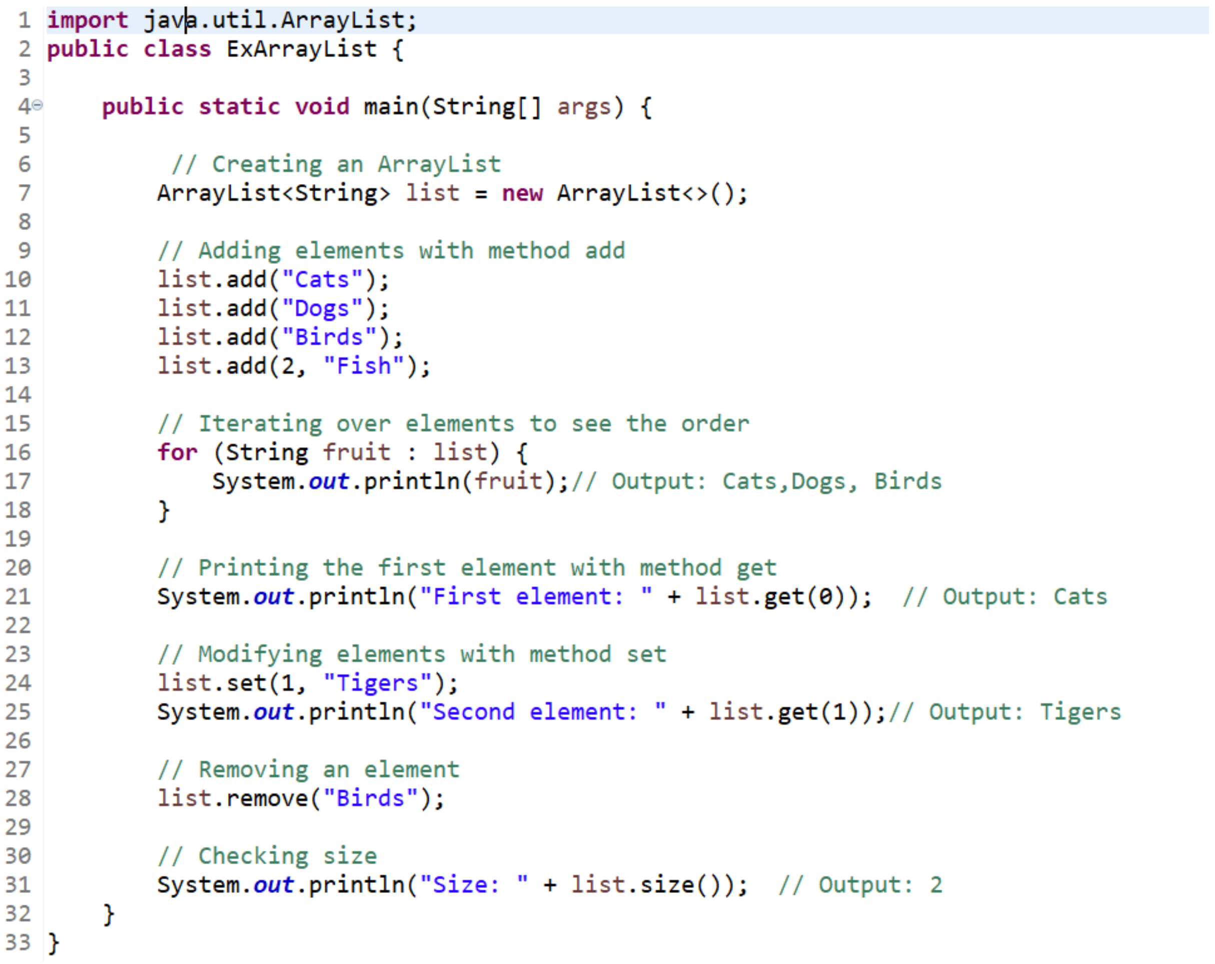
<u>ArrayList Class</u>
Key points:
- Implements: List, RandomAccess, Cloneable, Serializable
- It uses a dynamic array (can grow and shrink) to store the duplicate element of different data types.
- Maintains the insertion order and is non-synchronized.
- Maintains the order in which elements are inserted.
- Elements can be accessed in this order using an index or iterator.
- The RandomAccess interface is a marker interface used to indicate that the ArrayList supports fast (usually constant time) random access operations.

  Specific Methods:
    - *ensureCapacity(int minCapacity):* Increases the capacity of the ArrayList to ensure that it can hold at least the number of elements specified by the minimum capacity. This method is specific to ArrayList

and is used to optimize performance when you know in advance the number of elements you need to add.

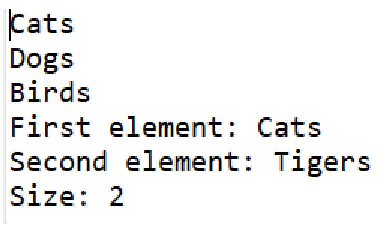
- *trimToSize():* Resizes the internal array to be the list’s current size. This can help to free up unused memory. This method is specific to ArrayList.

Example



```java
import java.util.ArrayList;
public class ExArrayList {

    public static void main(String[] args) {

         // Creating an ArrayList
        ArrayList<String> list = new ArrayList<>();

        // Adding elements with method add
        list.add("Cats");
        list.add("Dogs");
        list.add("Birds");
        list.add(2, "Fish");

        // Iterating over elements to see the order
        for (String fruit : list) {
            System.out.println(fruit);// Output: Cats,Dogs, Birds
        }

        // Printing the first element with method get
        System.out.println("First element: " + list.get(0));  // Output: Cats

        // Modifying elements with method set
        list.set(1, "Tigers");
        System.out.println("Second element: " + list.get(1));// Output: Tigers

        // Removing an element
        list.remove("Birds");

        // Checking size
        System.out.println("Size: " + list.size());  // Output: 2
    }
}
```

Output:



```
Cats
Dogs
Birds
First element: Cats
Second element: Tigers
Size: 2
```

<u>LinkedList Class</u>

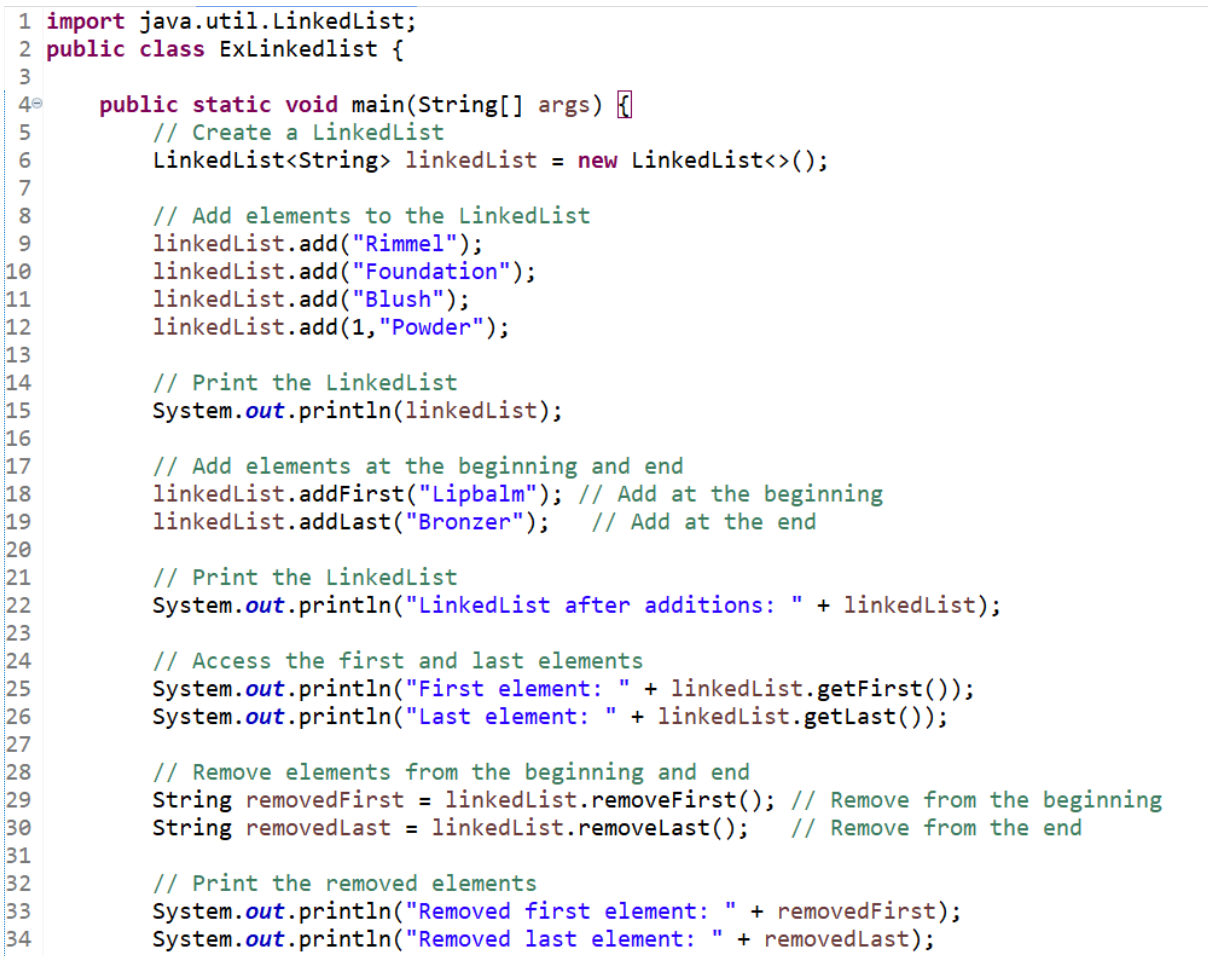
Key points:

- Implements List, Queue, and Dequeue interface.
- LinkedList Stores data and address.
- Double LinkedList Stores 3 different values in a single position, previous address, data and next address.

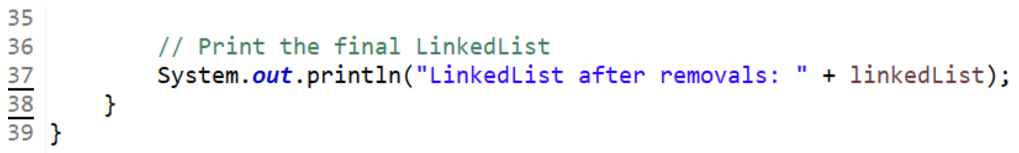
- When an element is added, previous and next addresses are changed.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- No random access of elements.
- Iteration needed from beginning of the List to access the element.
- It maintains the insertion order and is not synchronized.
- Allows storing null elements
- The manipulation is fast because no shifting is required.
- As it implements Deque, you can use it as both a queue (FIFO) and a stack (LIFO), making it flexible for various use cases.

  Methods:
  - *addFirst(E e)*: Inserts the specified element at the beginning of the list.
  - *addLast(E e):* Appends the specified element to the end of the list.
  - *getFirst():* Returns the first element in the list without removing it.
  - *getLast():* Returns the last element in the list without removing
  - *it.removeFirst():* Removes and returns the first element in the list.
  - *removeLast():* Removes and returns the last element in the list.
  - *peekFirst():* Retrieves, but does not remove, the first element of the list.
  - *peekLast():* Retrieves, but does not remove, the last element of the list.
  - *pollFirst():* Retrieves and removes the first element of the list, or returns null if the list is empty.
  - *pollLast()*: Retrieves and removes the last element of the list, or returns null if the list is empty.

Example:

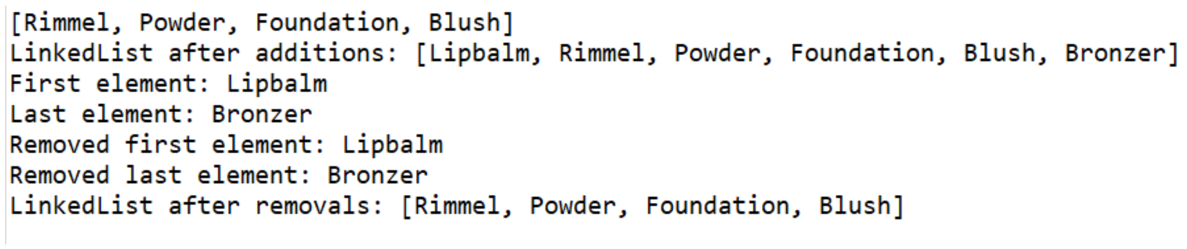


```java
import java.util.LinkedList;
public class ExLinkedlist {

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

        // Add elements to the LinkedList
        linkedList.add("Rimmel");
        linkedList.add("Foundation");
        linkedList.add("Blush");
        linkedList.add(1,"Powder");

        // Print the LinkedList
        System.out.println(linkedList);

        // Add elements at the beginning and end
        linkedList.addFirst("Lipbalm"); // Add at the beginning
        linkedList.addLast("Bronzer");   // Add at the end

        // Print the LinkedList
        System.out.println("LinkedList after additions: " + linkedList);

        // Access the first and last elements
        System.out.println("First element: " + linkedList.getFirst());
        System.out.println("Last element: " + linkedList.getLast());

        // Remove elements from the beginning and end
        String removedFirst = linkedList.removeFirst(); // Remove from the beginning
        String removedLast = linkedList.removeLast();   // Remove from the end

        // Print the removed elements
        System.out.println("Removed first element: " + removedFirst);
        System.out.println("Removed last element: " + removedLast);
```



```
        // Print the final LinkedList
        System.out.println("LinkedList after removals: " + linkedList);
    }
}
```

Output:



```
[Rimmel, Powder, Foundation, Blush]
LinkedList after additions: [Lipbalm, Rimmel, Powder, Foundation, Blush, Bronzer]
First element: Lipbalm
Last element: Bronzer
Removed first element: Lipbalm
Removed last element: Bronzer
LinkedList after removals: [Rimmel, Powder, Foundation, Blush]
```

<u>**Queue interface**</u>

Key points:

- It follows the FIFO (First-In-First-Out) principle, meaning elements are added to the end of the queue and removed from the front.
- There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
    - LinkedList: Implements Queue and provides a doubly linked list data structure.
    - PriorityQueue: Implements Queue and uses a priority heap to order elements.
    - ArrayDeque: Implements Queue using a resizable array, providing fast access and modification.
    - Deque: Extends the Queue interface. Deque stands for "Double-Ended Queue," and it provides methods for inserting, removing, and accessing elements at both ends of the queue.

  Common Methods:
    - boolean add(E e): Inserts the specified element into the queue. Throws an IllegalStateException if the queue is full.
    - boolean offer(E e): Inserts the specified element into the queue if possible. Returns true if the element was added successfully, or false if the queue is full (in bounded queues).
    - E remove(): Retrieves and removes the head of the queue. Throws a NoSuchElementException if the queue is empty.
    - E poll(): Retrieves and removes the head of the queue, or returns null if the queue is empty.
    - E peek(): Retrieves, but does not remove, the head of the queue. Returns null if the queue is empty.
    - E element(): Retrieves, but does not remove, the head of the queue. Throws a NoSuchElementException if the queue is empty.

- int size(): Returns the number of elements in the queue.
- boolean isEmpty(): Returns true if the queue contains no elements.
- boolean contains(Object o): Returns true if the queue contains the specified element.

PriorityQueue Class

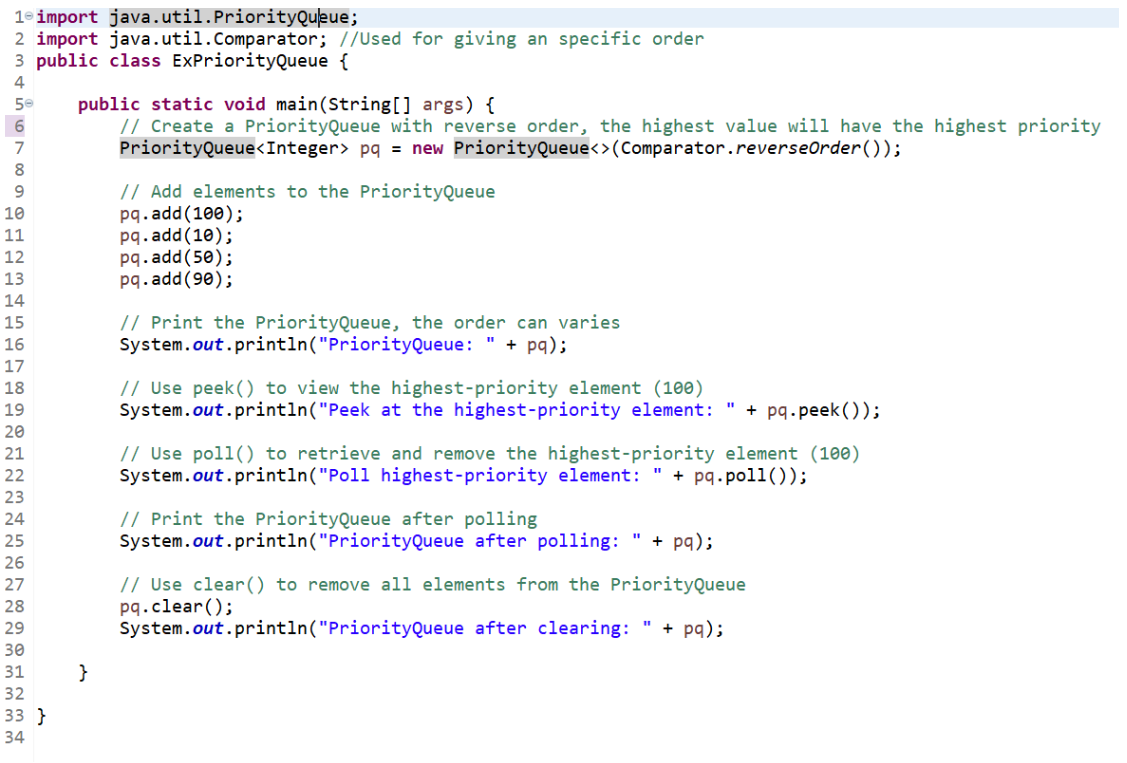
Key points:

- Does not follow the FIFO (First-In-First-Out) principle; instead, it orders elements based on their priority.
- Orders its elements according to their natural ordering or by a specified comparator.
- Does not allow null values to be stored in the queue.
- Does not have a fixed capacity. It grows dynamically as elements are added.
- We can’t create a PriorityQueue of Objects that are non-comparable.
- It inherits methods from AbstractQueue, AbstractCollection, Collection, and Object class.

Methods:

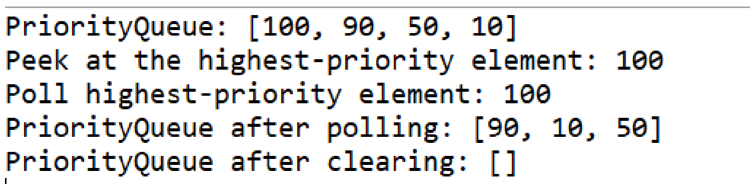
- E peek(): Retrieves, but does not remove, the highest-priority element in the queue. Returns null if the queue is empty.
- E poll(): Retrieves and removes the highest-priority element from the queue. Returns null if the queue is empty.
- void clear(): Removes all elements from the priority queue. This method is inherited from the Collection interface but is significant for managing priority queues.
- boolean remove(Object o): Removes a single instance of the specified element from the queue, if it is present. Also inherited from the Collection interface.

Example:



```java
import java.util.PriorityQueue;
import java.util.Comparator; //Used for giving an specific order
public class ExPriorityQueue {

    public static void main(String[] args) {
        // Create a PriorityQueue with reverse order, the highest value will have the highest priority
        PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());

        // Add elements to the PriorityQueue
        pq.add(100);
        pq.add(10);
        pq.add(50);
        pq.add(90);

        // Print the PriorityQueue, the order can varies
        System.out.println("PriorityQueue: " + pq);

        // Use peek() to view the highest-priority element (100)
        System.out.println("Peek at the highest-priority element: " + pq.peek());

        // Use poll() to retrieve and remove the highest-priority element (100)
        System.out.println("Poll highest-priority element: " + pq.poll());

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

        // Use clear() to remove all elements from the PriorityQueue
        pq.clear();
        System.out.println("PriorityQueue after clearing: " + pq);

    }

}
```

Output:



```
PriorityQueue: [100, 90, 50, 10]
Peek at the highest-priority element: 100
Poll highest-priority element: 100
PriorityQueue after polling: [90, 10, 50]
PriorityQueue after clearing: []
```

<u>Deque Interface</u>

Key points:

- Insertion and Removal: Allows adding and removing elements from both the front and the rear of the queue.
- Access: Provides methods to inspect elements from both ends.
- Doesn´t accept null elements.

  Methods:

  - void addFirst(E e): Inserts the specified element at the front of the deque.
  - void addLast(E e): Inserts the specified element at the end of the deque.
  - boolean offerFirst(E e): Inserts the specified element at the front of the deque if possible. Returns true if successful, or false if the deque is full.
  - boolean offerLast(E e): Inserts the specified element at the end of the deque if possible. Returns true if successful, or false if the deque is full.
  - E removeFirst(): Removes and returns the element at the front of the deque. Throws NoSuchElementException if the deque is empty.
  - E removeLast(): Removes and returns the element at the end of the deque. Throws NoSuchElementException if the deque is empty.
  - E pollFirst(): Retrieves and removes the element at the front of the deque, or returns null if the deque is empty.
  - E pollLast(): Retrieves and removes the element at the end of the deque, or returns null if the deque is empty.
  - E getFirst(): Retrieves, but does not remove, the element at the front of the deque. Throws NoSuchElementException if the deque is empty.
  - E getLast(): Retrieves, but does not remove, the element at the end of the deque. Throws NoSuchElementException if the deque is empty.
  - E peekFirst(): Retrieves, but does not remove, the element at the front of the deque, or returns null if the deque is empty.
  - E peekLast(): Retrieves, but does not remove, the element at the end of the deque, or returns null if the deque is empty.

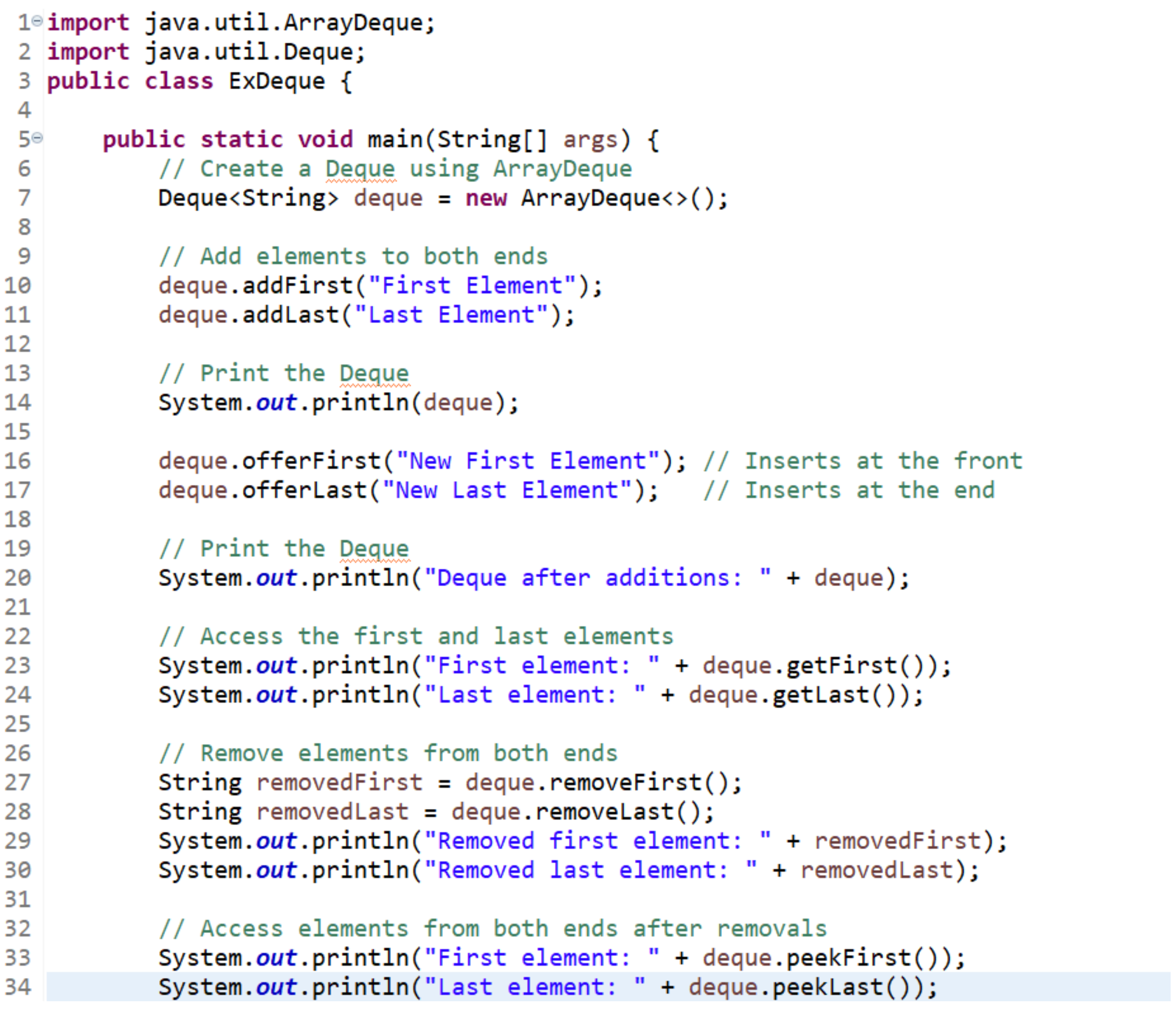
<u>ArrayDeque Class</u>

- Array Double-Ended Queue allows elements to be added or removed from both ends (front and back). This makes it a versatile structure for implementing both stacks and queues.

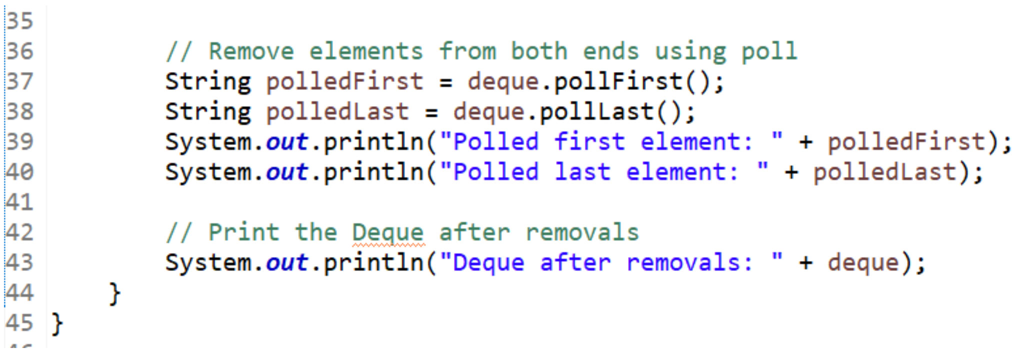
- Is a concrete class that provides an implementation of the Deque interface using a resizable array.
- Does not have a fixed capacity. It automatically resizes when necessary, offering dynamic array-based storage.
- Does not permit null elements.
- Is not synchronized, meaning it is not thread-safe by default. External synchronization is needed if multiple threads are accessing the deque concurrently.
- Does not support random access to elements. Elements are accessed in a sequential manner, either from the front or the back.

  Methods:
  - ArrayDeque in Java implements the Deque interface, so it inherits all methods from Deque.
  - push(E e): Pushes an element onto the stack represented by this deque (equivalent to addFirst(E e)).
  - pop(): Pops an element from the stack represented by this deque (equivalent to removeFirst()).

Example:

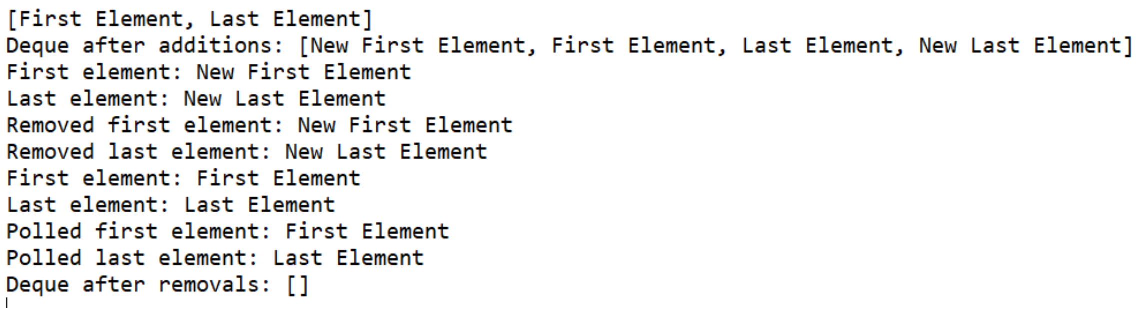


```java
import java.util.ArrayDeque;
import java.util.Deque;
public class ExDeque {

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

        // Add elements to both ends
        deque.addFirst("First Element");
        deque.addLast("Last Element");

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

        deque.offerFirst("New First Element"); // Inserts at the front
        deque.offerLast("New Last Element");   // Inserts at the end

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

        // Access the first and last elements
        System.out.println("First element: " + deque.getFirst());
        System.out.println("Last element: " + deque.getLast());

        // Remove elements from both ends
        String removedFirst = deque.removeFirst();
        String removedLast = deque.removeLast();
        System.out.println("Removed first element: " + removedFirst);
        System.out.println("Removed last element: " + removedLast);

        // Access elements from both ends after removals
        System.out.println("First element: " + deque.peekFirst());
        System.out.println("Last element: " + deque.peekLast());
```



```java

        // Remove elements from both ends using poll
        String polledFirst = deque.pollFirst();
        String polledLast = deque.pollLast();
        System.out.println("Polled first element: " + polledFirst);
        System.out.println("Polled last element: " + polledLast);

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

Output:



```
[First Element, Last Element]
Deque after additions: [New First Element, First Element, Last Element, New Last Element]
First element: New First Element
Last element: New Last Element
Removed first element: New First Element
Removed last element: New Last Element
First element: First Element
Last element: Last Element
Polled first element: First Element
Polled last element: Last Element
Deque after removals: []
```

<u>**Set Interface**</u>

Key points:

- A Set does not allow duplicate elements.
- Does not maintain any particular order of its elements. The order in which elements are stored is often dependent on the specific Set implementation used.
- Is an interface that extends the Collection interface. Therefore, it inherits all the methods from Collection like add(), remove(), contains(), size(), isEmpty(), and more.
- The classes that implement the Set interface are HashSet, LinkedHashSet, and TreeSet.

  Common Methods:

  - add(E e): Adds the specified element to the set if it is not already present. Returns true if the set did not already contain the specified element.
  - addAll(Collection<? extends E> c): Adds all of the elements in the specified collection to the set if they're not already present. Returns true if the set changed as a result of the operation.
  - clear(): Removes all elements from the set, leaving it empty.
  - contains(Object o): Returns true if the set contains the specified element.

  - containsAll(Collection<?> c): Returns true if the set contains all the elements in the specified collection.
  - equals(Object o): Compares the specified object with the set for equality. Returns true if the specified object is equal to the set.
  - isEmpty(): Returns true if the set contains no elements.
  - iterator(): Returns an iterator over the elements in the set, allowing for iteration through the set.
  - remove(Object o): Removes the specified element from the set if it is present. Returns true if the set contained the specified element.
  - size(): Returns the number of elements in the set.
  - toArray(): Returns an array containing all of the elements in the set.

HashSet Class

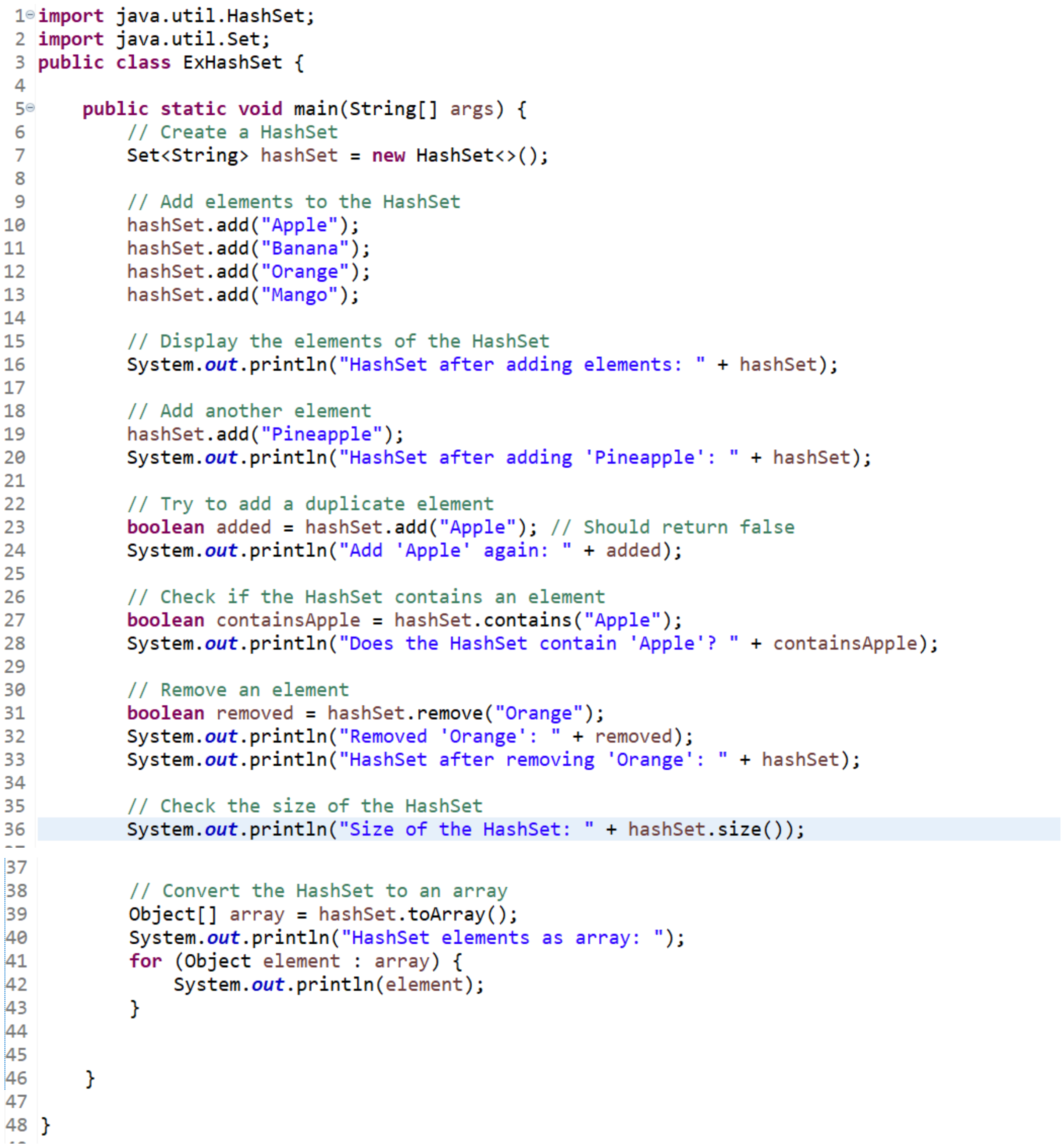
Key points:

- Implements the Set interface.
- Does not allow duplicate elements.
- Allows a single null element. If you add multiple null values, only one will be retained.
- Internally, HashSet uses a HashMap to store its elements. The elements are stored as keys in the HashMap, with a constant placeholder value.
- HashSet is not synchronized. If multiple threads access a HashSet concurrently, and at least one of the threads modifies the set, it must be synchronized externally.
- The order of iteration in a HashSet is not guaranteed.
- Is commonly used in scenarios where fast lookup, addition, and removal of elements are required, and where duplicates are not allowed.

  Methods:

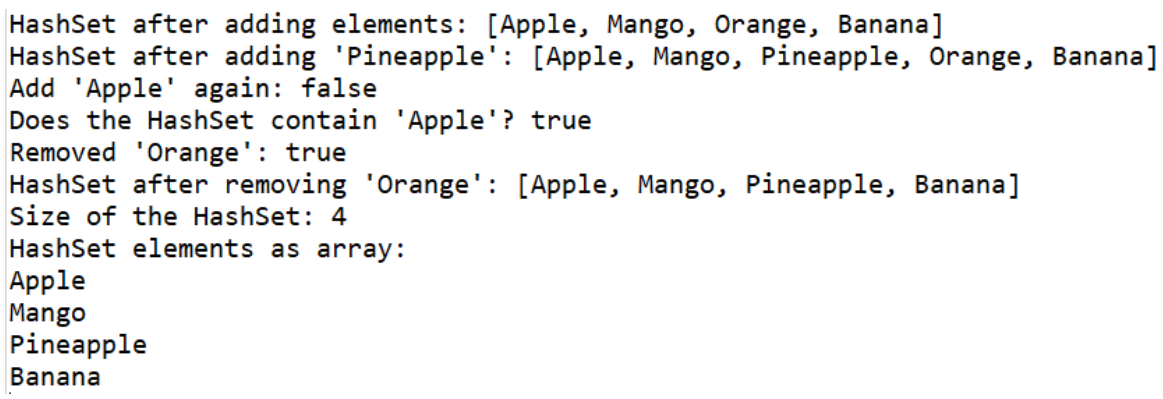
  - it inherits all its methods from the Set interface and the AbstractSet class. Since HashSet is a specific implementation of the Set interface, it does not add any new methods beyond those defined in Set.

Example:



```java
import java.util.HashSet;
import java.util.Set;
public class ExHashSet {

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

        // Add elements to the HashSet
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Orange");
        hashSet.add("Mango");

        // Display the elements of the HashSet
        System.out.println("HashSet after adding elements: " + hashSet);

        // Add another element
        hashSet.add("Pineapple");
        System.out.println("HashSet after adding 'Pineapple': " + hashSet);

        // Try to add a duplicate element
        boolean added = hashSet.add("Apple"); // Should return false
        System.out.println("Add 'Apple' again: " + added);

        // Check if the HashSet contains an element
        boolean containsApple = hashSet.contains("Apple");
        System.out.println("Does the HashSet contain 'Apple'? " + containsApple);

        // Remove an element
        boolean removed = hashSet.remove("Orange");
        System.out.println("Removed 'Orange': " + removed);
        System.out.println("HashSet after removing 'Orange': " + hashSet);

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

        // Convert the HashSet to an array
        Object[] array = hashSet.toArray();
        System.out.println("HashSet elements as array: ");
        for (Object element : array) {
            System.out.println(element);
        }


    }

}
```

Output:



```
HashSet after adding elements: [Apple, Mango, Orange, Banana]
HashSet after adding 'Pineapple': [Apple, Mango, Pineapple, Orange, Banana]
Add 'Apple' again: false
Does the HashSet contain 'Apple'? true
Removed 'Orange': true
HashSet after removing 'Orange': [Apple, Mango, Pineapple, Banana]
Size of the HashSet: 4
HashSet elements as array:
Apple
Mango
Pineapple
Banana
```

<u>LinkedHashSet Class</u>

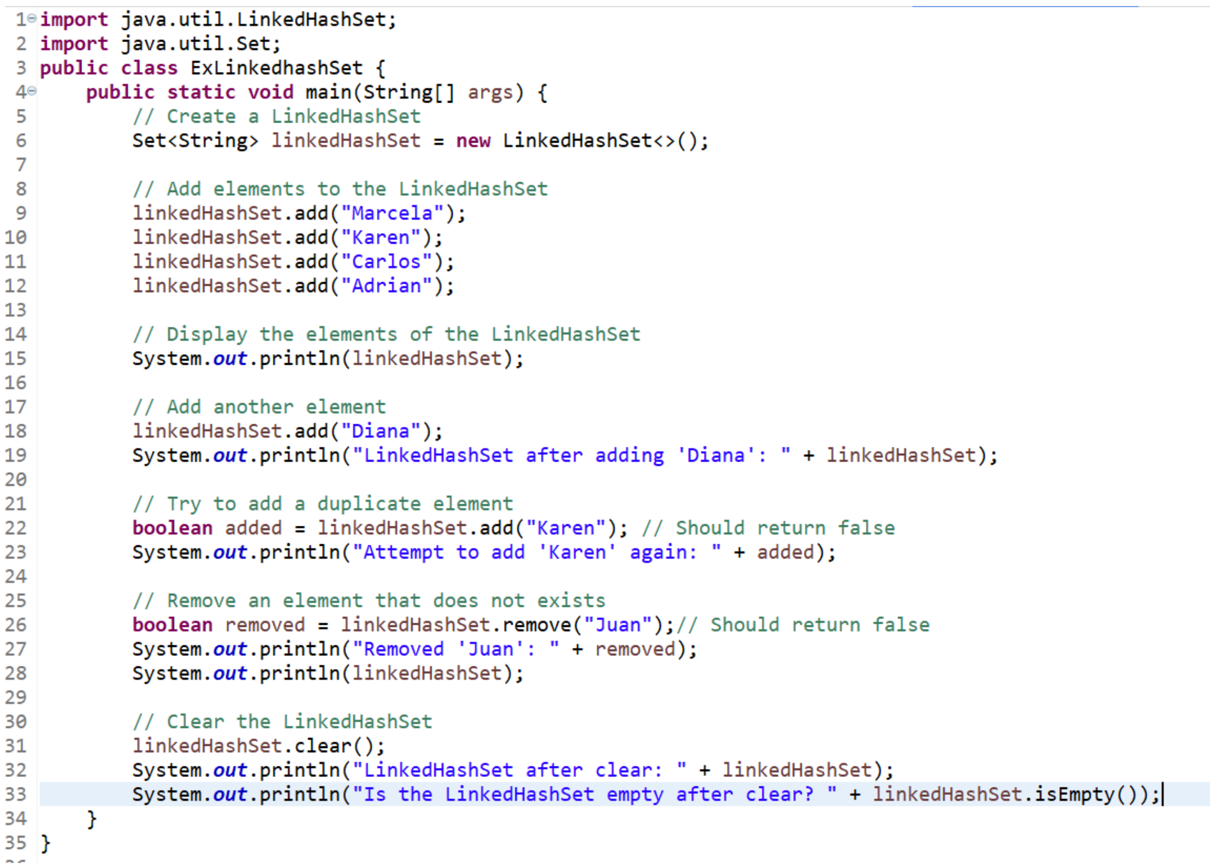
Key points:

- Implements the Set interface, inheriting methods from Set and AbstractSet.
- Maintains the order of elements as they were inserted. This is achieved using a linked list that keeps track of the insertion order.
- Does not allow duplicate elements.
- Allows a single null element.
- Internally, LinkedHashSet uses a hash table (like HashSet) to store elements and a doubly-linked list to maintain the insertion order.
- Is not synchronized. If multiple threads access a LinkedHashSet concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

  Methods:

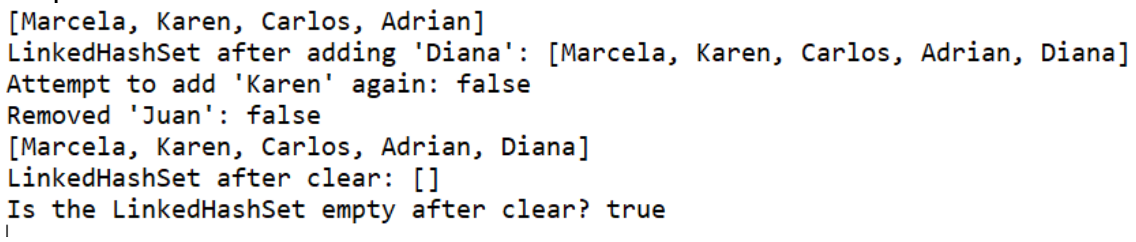
  - LinkedHashSet does not have exclusive methods beyond those provided by HashSet and AbstractSet. Its unique features are related to maintaining the order of elements rather than introducing new methods.

Example:



```java
import java.util.LinkedHashSet;
import java.util.Set;
public class ExLinkedhashSet {
    public static void main(String[] args) {
        // Create a LinkedHashSet
        Set<String> linkedHashSet = new LinkedHashSet<>();

        // Add elements to the LinkedHashSet
        linkedHashSet.add("Marcela");
        linkedHashSet.add("Karen");
        linkedHashSet.add("Carlos");
        linkedHashSet.add("Adrian");

        // Display the elements of the LinkedHashSet
        System.out.println(linkedHashSet);

        // Add another element
        linkedHashSet.add("Diana");
        System.out.println("LinkedHashSet after adding 'Diana': " + linkedHashSet);

        // Try to add a duplicate element
        boolean added = linkedHashSet.add("Karen"); // Should return false
        System.out.println("Attempt to add 'Karen' again: " + added);

        // Remove an element that does not exists
        boolean removed = linkedHashSet.remove("Juan");// Should return false
        System.out.println("Removed 'Juan': " + removed);
        System.out.println(linkedHashSet);

        // Clear the LinkedHashSet
        linkedHashSet.clear();
        System.out.println("LinkedHashSet after clear: " + linkedHashSet);
        System.out.println("Is the LinkedHashSet empty after clear? " + linkedHashSet.isEmpty());
    }
}
```

Output:



```
[Marcela, Karen, Carlos, Adrian]
LinkedHashSet after adding 'Diana': [Marcela, Karen, Carlos, Adrian, Diana]
Attempt to add 'Karen' again: false
Removed 'Juan': false
[Marcela, Karen, Carlos, Adrian, Diana]
LinkedHashSet after clear: []
Is the LinkedHashSet empty after clear? true
```

SortedSet Interface

Key points:

- The SortedSet interface in Java extends the Set interface and provides additional methods for dealing with sorted collections.
- Elements in a SortedSet are ordered based on their natural ordering (if they implement Comparable) or according to a Comparator provided at set creation time.
- Does not allow duplicate objects.
- The elements of the SortedSet are arranged in the increasing (ascending) order.
- Common implementations of SortedSet include TreeSet, which provides log(n) time complexity for most operations, due to its underlying Red-Black Tree structure.

  Common Methods:

  - Inherits all the methods from the Set interface
  - first(): Returns the first (lowest) element in the set.
  - last(): Returns the last (highest) element in the set.
  - headSet(E toElement): Returns a view of the portion of the set whose elements are strictly less than toElement.
  - tailSet(E fromElement): Returns a view of the portion of the set whose elements are greater than or equal to fromElement.
  - subSet(E fromElement, E toElement): Returns a view of the portion of the set between fromElement (inclusive) and toElement (exclusive).
  - comparator(): Returns the comparator used to sort the elements of the set. If the set is sorted according to natural ordering, it returns null.

TreeSet Class

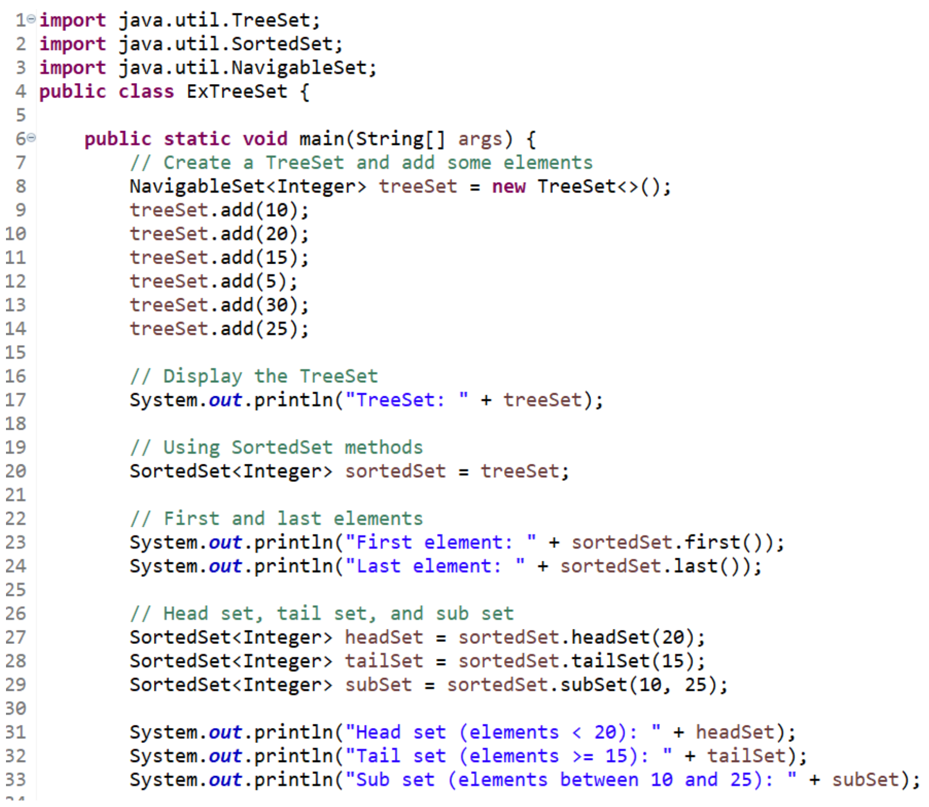
- Implements the SortedSet and NavigableSet interfaces, providing methods for sorted and navigable set operations.
- Elements in a TreeSet are stored in a sorted order. The set is sorted according to their natural ordering (if they implement Comparable) or by a Comparator provided at the time of creation.
- Does not allow duplicate elements. Each element must be unique, determined by the equals() method.
- Does not allow null elements. Attempting to add null will result in a NullPointerException.
- Iterators returned by TreeSet traverse the elements in ascending order.
- It supports navigational methods for more advanced querying and operations, and offers various constructors for flexibility in creation.

  Methods:

  - TreeSet inherits methods from the SortedSet and NavigableSet interfaces, but it does not have exclusive methods beyond those provided by these interfaces. The following methods are from NavigableSet.

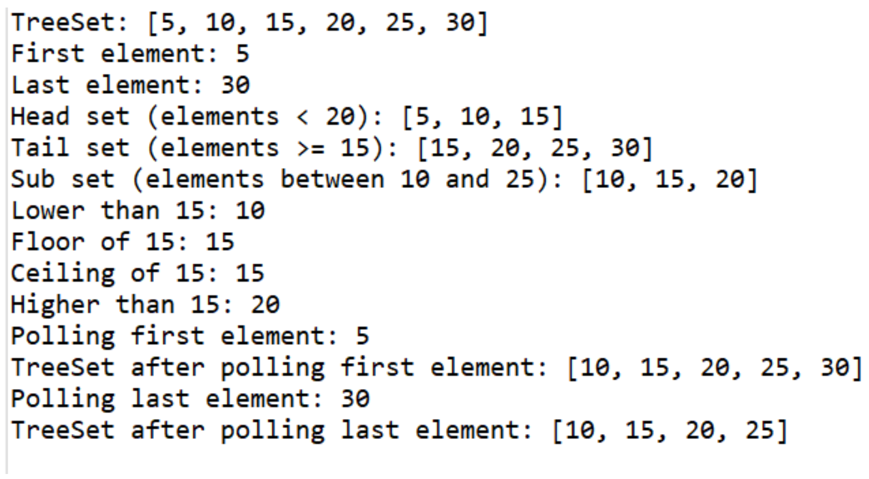
- lower(E e): Returns the greatest element less than the specified element, or null if there is no such element.
- floor(E e): Returns the greatest element less than or equal to the specified element, or null if there is no such element.
- ceiling(E e): Returns the least element greater than or equal to the specified element, or null if there is no such element.
- higher(E e): Returns the least element greater than the specified element, or null if there is no such element.
- pollFirst(): Retrieves and removes the first (lowest) element, or returns null if the set is empty.
- pollLast(): Retrieves and removes the last (highest) element, or returns null if the set is empty.
- descendingIterator(): Returns an iterator over the elements in this set, in descending order.

Example:



```java
import java.util.TreeSet;
import java.util.SortedSet;
import java.util.NavigableSet;
public class ExTreeSet {

    public static void main(String[] args) {
        // Create a TreeSet and add some elements
        NavigableSet<Integer> treeSet = new TreeSet<>();
        treeSet.add(10);
        treeSet.add(20);
        treeSet.add(15);
        treeSet.add(5);
        treeSet.add(30);
        treeSet.add(25);

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

        // Using SortedSet methods
        SortedSet<Integer> sortedSet = treeSet;

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

        // Head set, tail set, and sub set
        SortedSet<Integer> headSet = sortedSet.headSet(20);
        SortedSet<Integer> tailSet = sortedSet.tailSet(15);
        SortedSet<Integer> subSet = sortedSet.subSet(10, 25);

        System.out.println("Head set (elements < 20): " + headSet);
        System.out.println("Tail set (elements >= 15): " + tailSet);
        System.out.println("Sub set (elements between 10 and 25): " + subSet);
```

Output:



```
TreeSet: [5, 10, 15, 20, 25, 30]
First element: 5
Last element: 30
Head set (elements < 20): [5, 10, 15]
Tail set (elements >= 15): [15, 20, 25, 30]
Sub set (elements between 10 and 25): [10, 15, 20]
Lower than 15: 10
Floor of 15: 15
Ceiling of 15: 15
Higher than 15: 20
Polling first element: 5
TreeSet after polling first element: [10, 15, 20, 25, 30]
Polling last element: 30
TreeSet after polling last element: [10, 15, 20, 25]
```

<u>Map Interface</u>

Key points:

- A Map stores data as key-value pairs. Each key maps to exactly one value. Keys must be unique, but values can be duplicated.
- Some Map implementations allow null keys and values (e.g., HashMap). Others, like TreeMap, do not allow null keys but may allow null values.
- A Map is useful if you have to search, update or delete elements on the basis of a key.
- There are three classes to implement maps: HashMap, LinkedHashMap, and TreeMap.

  Methods:
  - put(K key, V value): Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
  - 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 a key from this map if it is 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.
  - 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, where each mapping is represented by a Map.Entry.
  - 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.
- putAll(Map<? extends K, ? extends V> m): Copies all of the mappings from the specified map to this map.
- equals(Object o): Compares the specified object with this map for equality.
- hashCode(): Returns the hash code value for this map.

HashMap Class

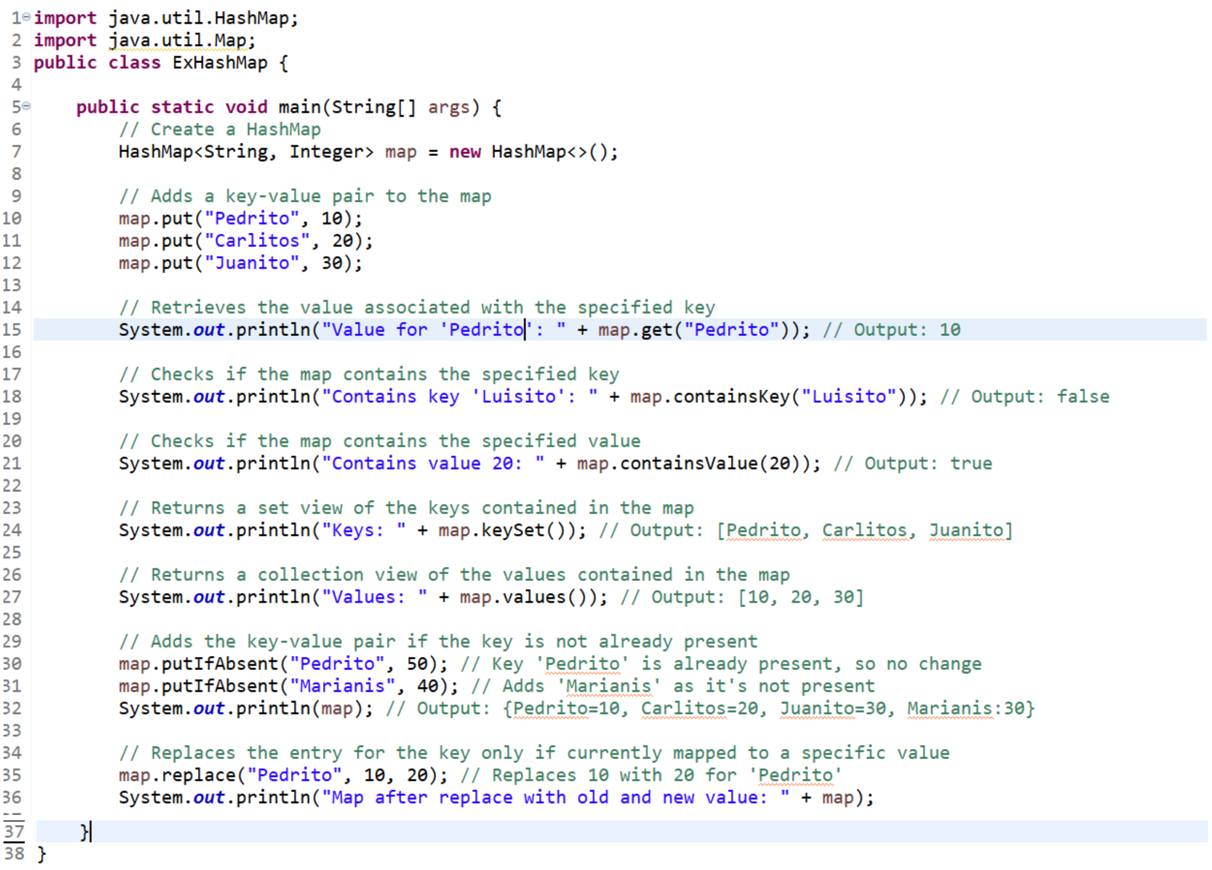
Key Points:

- Implements the Map interface, which means it stores data as key-value pairs.
- It doesn't maintain any order.
- Allows one null key and multiple null values. However, it can only have one null key since keys must be unique.
- The iteration order of the keys is based on the hash codes of the keys and can change over time.
- When the number of entries in the HashMap exceeds the product of the load factor and the current capacity, the capacity is increased and the existing entries are rehashed to the new buckets.
- Is not synchronized, meaning it is not thread-safe.

  Methods:
    - Inherits methods from Map.
    - putIfAbsent(K key, V value): Inserts the specified key-value pair into the map only if the key is not already associated with a value.
    - remove(Object key, Object value): Removes the entry for the specified key only if it is currently mapped to the specified value.
    - replace(K key, V oldValue, V newValue): Replaces the entry for the specified key only if it is currently mapped to the specified old value.
    - replace(K key, V value): Replaces the entry for the specified key only if it is currently mapped to some value.
    - compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction): Attempts to compute a new mapping for the specified key and its current mapped value (or null if there is no current mapping).
    - computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction): If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into the map.
    - computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction): If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
    - merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction): If the specified key is not already associated with

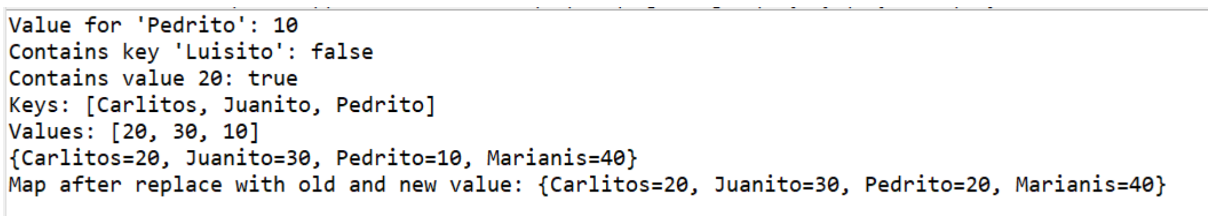
a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the value with the result of the given remapping function.

Example:



```java
import java.util.HashMap;
import java.util.Map;
public class ExHashMap {

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

        // Adds a key-value pair to the map
        map.put("Pedrito", 10);
        map.put("Carlitos", 20);
        map.put("Juanito", 30);

        // Retrieves the value associated with the specified key
        System.out.println("Value for 'Pedrito': " + map.get("Pedrito")); // Output: 10

        // Checks if the map contains the specified key
        System.out.println("Contains key 'Luisito': " + map.containsKey("Luisito")); // Output: false

        // Checks if the map contains the specified value
        System.out.println("Contains value 20: " + map.containsValue(20)); // Output: true

        // Returns a set view of the keys contained in the map
        System.out.println("Keys: " + map.keySet()); // Output: [Pedrito, Carlitos, Juanito]

        // Returns a collection view of the values contained in the map
        System.out.println("Values: " + map.values()); // Output: [10, 20, 30]

        // Adds the key-value pair if the key is not already present
        map.putIfAbsent("Pedrito", 50); // Key 'Pedrito' is already present, so no change
        map.putIfAbsent("Marianis", 40); // Adds 'Marianis' as it's not present
        System.out.println(map); // Output: {Pedrito=10, Carlitos=20, Juanito=30, Marianis:30}

        // Replaces the entry for the key only if currently mapped to a specific value
        map.replace("Pedrito", 10, 20); // Replaces 10 with 20 for 'Pedrito'
        System.out.println("Map after replace with old and new value: " + map);

    }
}
```

Output:



```
Value for 'Pedrito': 10
Contains key 'Luisito': false
Contains value 20: true
Keys: [Carlitos, Juanito, Pedrito]
Values: [20, 30, 10]
{Carlitos=20, Juanito=30, Pedrito=10, Marianis=40}
Map after replace with old and new value: {Carlitos=20, Juanito=30, Pedrito=20, Marianis=40}
```

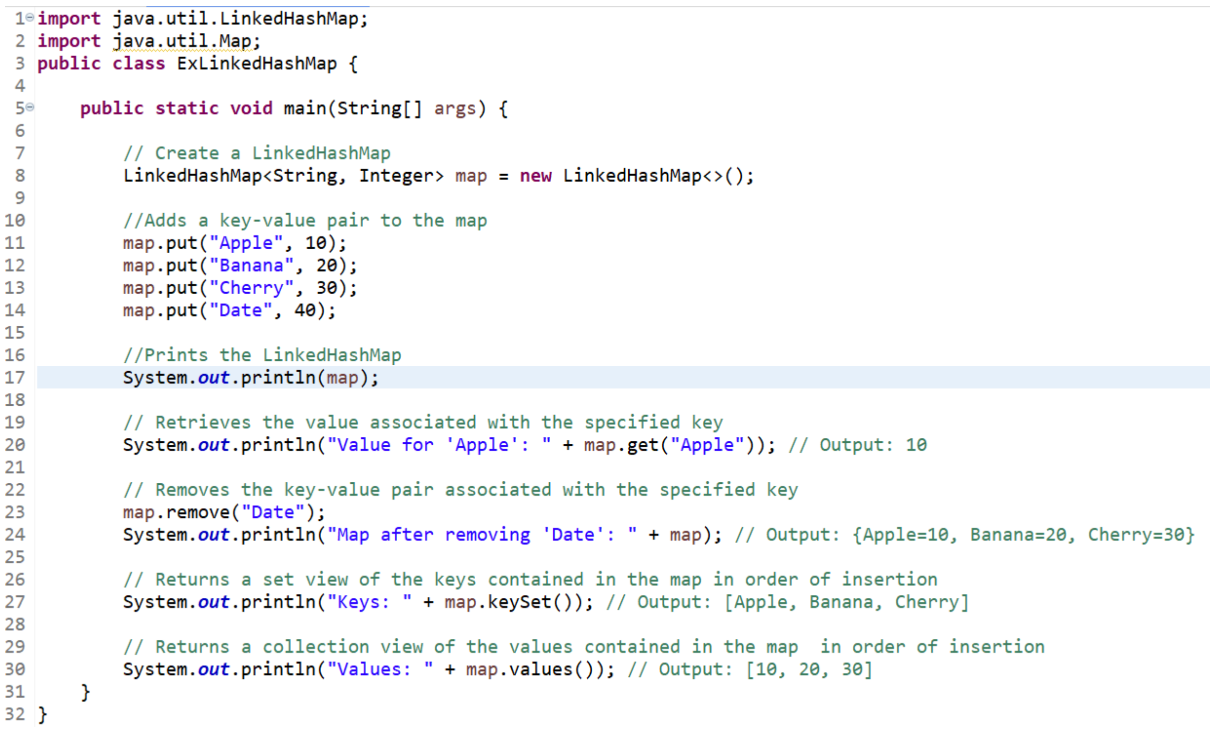
LinkedHashMap Class

Key points:

- Implements the HashMap class and Map interface.
- It maintains insertion order.
- Maintains a doubly-linked list of its entries. This linked list defines the iteration ordering, which is the order in which keys were inserted into the map
- Maintain entries based on access order (the order in which entries were last accessed, from least-recently accessed to most-recently accessed) if constructed with accessOrder = true. This is useful for implementing caches.
- Allows one null key and multiple null values.
- Is not synchronized, meaning it is not thread-safe.

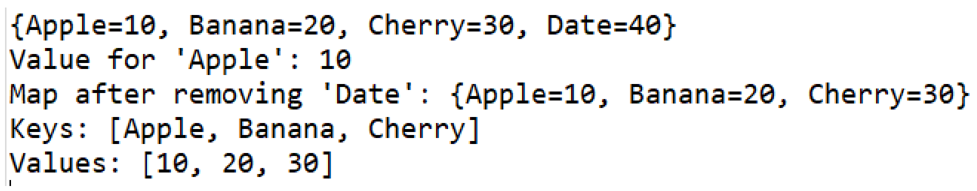
  Methods:
  - Inherits methods from HashMap and Map.
  - protected boolean removeEldestEntry(Map.Entry<K,V> eldest): This method is used to determine whether the eldest entry should be removed from the map when a new entry is added. By default, it returns false, meaning the eldest entry is not removed. However, you can override this method to return true when the size of the map exceeds a certain threshold, effectively implementing a Least Recently Used (LRU) cache.

Example:



```java
import java.util.LinkedHashMap;
import java.util.Map;
public class ExLinkedHashMap {

    public static void main(String[] args) {

        // Create a LinkedHashMap
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>();

        //Adds a key-value pair to the map
        map.put("Apple", 10);
        map.put("Banana", 20);
        map.put("Cherry", 30);
        map.put("Date", 40);

        //Prints the LinkedHashMap
        System.out.println(map);

        // Retrieves the value associated with the specified key
        System.out.println("Value for 'Apple': " + map.get("Apple")); // Output: 10

        // Removes the key-value pair associated with the specified key
        map.remove("Date");
        System.out.println("Map after removing 'Date': " + map); // Output: {Apple=10, Banana=20, Cherry=30}

        // Returns a set view of the keys contained in the map in order of insertion
        System.out.println("Keys: " + map.keySet()); // Output: [Apple, Banana, Cherry]

        // Returns a collection view of the values contained in the map  in order of insertion
        System.out.println("Values: " + map.values()); // Output: [10, 20, 30]
    }
}
```

Output:



```
{Apple=10, Banana=20, Cherry=30, Date=40}
Value for 'Apple': 10
Map after removing 'Date': {Apple=10, Banana=20, Cherry=30}
Keys: [Apple, Banana, Cherry]
Values: [10, 20, 30]
```

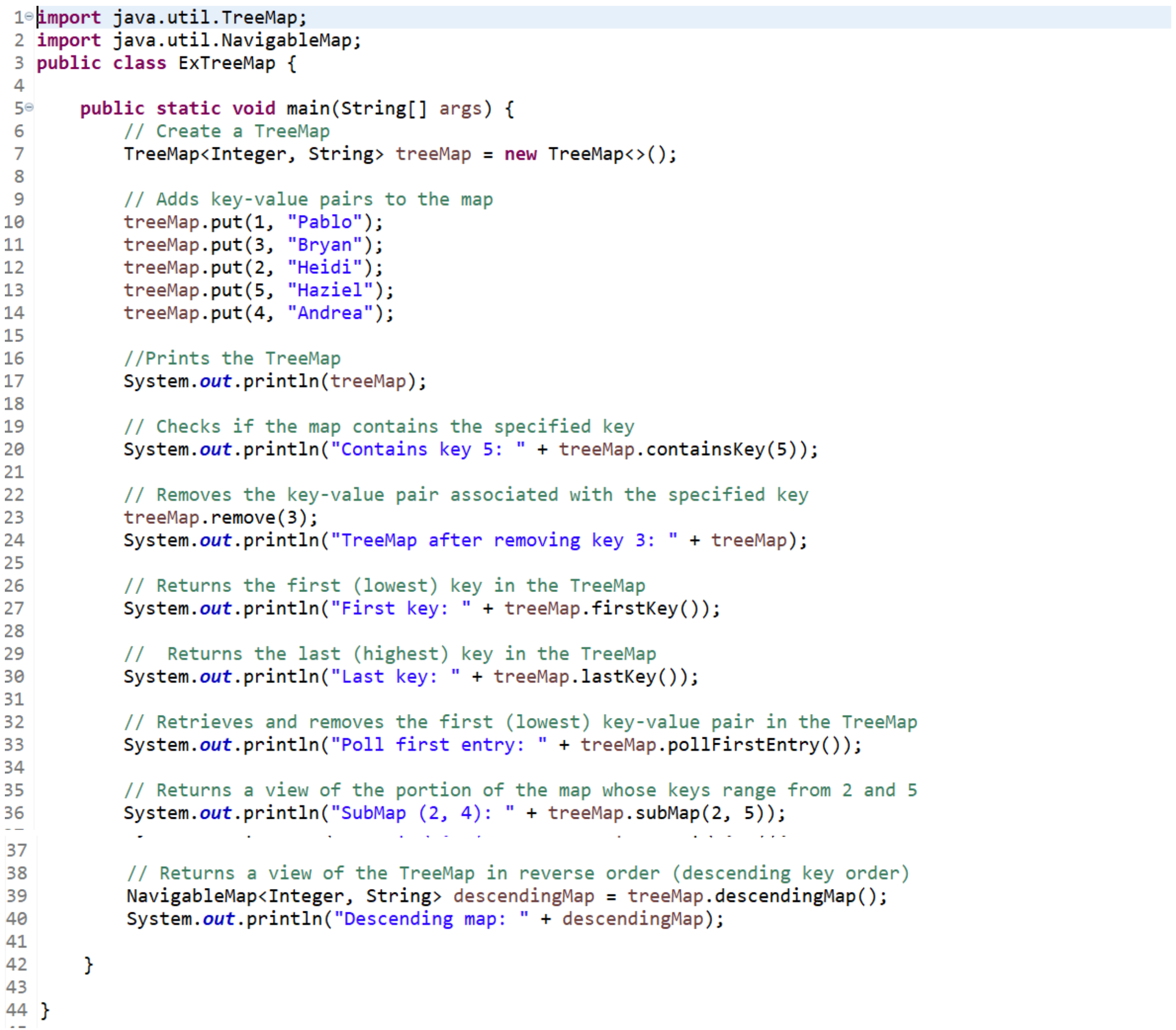
TreeMap Class

Key Points:

- Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- Contains only unique elements.
- The keys in a TreeMap are sorted according to their natural ordering (ascending order) or by a Comparator provided at map creation time. By default, it uses the natural ordering of the keys.
- Does not allow null keys. However, null values are allowed.
- Is not synchronized. For thread-safe operations, you need to use Collections.synchronizedMap or another synchronization mechanism.

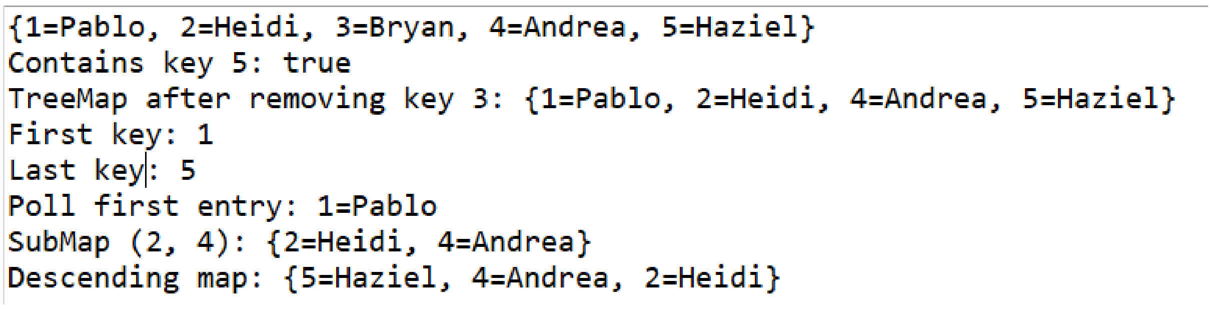
  Methods:
    - firstKey(): Returns the first (lowest) key in the TreeMap.
    - lastKey(): Returns the last (highest) key in the TreeMap.
    - pollFirstEntry(): Retrieves and removes the first (lowest) key-value pair in the TreeMap. Returns null if the map is empty.
    - pollLastEntry(): Retrieves and removes the last (highest) key-value pair in the TreeMap. Returns null if the map is empty.
    - subMap(K fromKey, K toKey): Returns a view of the portion of the map whose keys range from fromKey (inclusive) to toKey (exclusive).
    - headMap(K toKey): Returns a view of the portion of the map whose keys are less than toKey.
    - tailMap(K fromKey): Returns a view of the portion of the map whose keys are greater than or equal to fromKey.
    - descendingMap(): Returns a view of the TreeMap in reverse order (descending key order).
    - ceilingEntry(K key): Returns a key-value pair associated with the least key greater than or equal to key, or null if no such key exists.
    - floorEntry(K key): Returns a key-value pair associated with the greatest key less than or equal to key, or null if no such key exists.
    - higherEntry(K key): Returns a key-value pair associated with the least key strictly greater than key, or null if no such key exists.
    - lowerEntry(K key): Returns a key-value pair associated with the greatest key strictly less than key, or null if no such key exists.

Example:



```java
import java.util.TreeMap;
import java.util.NavigableMap;
public class ExTreeMap {

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

        // Adds key-value pairs to the map
        treeMap.put(1, "Pablo");
        treeMap.put(3, "Bryan");
        treeMap.put(2, "Heidi");
        treeMap.put(5, "Haziel");
        treeMap.put(4, "Andrea");

        //Prints the TreeMap
        System.out.println(treeMap);

        // Checks if the map contains the specified key
        System.out.println("Contains key 5: " + treeMap.containsKey(5));

        // Removes the key-value pair associated with the specified key
        treeMap.remove(3);
        System.out.println("TreeMap after removing key 3: " + treeMap);

        // Returns the first (lowest) key in the TreeMap
        System.out.println("First key: " + treeMap.firstKey());

        //  Returns the last (highest) key in the TreeMap
        System.out.println("Last key: " + treeMap.lastKey());

        // Retrieves and removes the first (lowest) key-value pair in the TreeMap
        System.out.println("Poll first entry: " + treeMap.pollFirstEntry());

        // Returns a view of the portion of the map whose keys range from 2 and 5
        System.out.println("SubMap (2, 4): " + treeMap.subMap(2, 5));

        // Returns a view of the TreeMap in reverse order (descending key order)
        NavigableMap<Integer, String> descendingMap = treeMap.descendingMap();
        System.out.println("Descending map: " + descendingMap);

    }

}
```

Output:



```
{1=Pablo, 2=Heidi, 3=Bryan, 4=Andrea, 5=Haziel}
Contains key 5: true
TreeMap after removing key 3: {1=Pablo, 2=Heidi, 4=Andrea, 5=Haziel}
First key: 1
Last key: 5
Poll first entry: 1=Pablo
SubMap (2, 4): {2=Heidi, 4=Andrea}
Descending map: {5=Haziel, 4=Andrea, 2=Heidi}
```

Conclusion

In conclusion,the Java Collections Framework is packed with different interfaces and classes to help you manage and work with data. Interfaces like Set, List, Queue, and Deque outline how you should interact with collections, while classes such as ArrayList, LinkedList, PriorityQueue, HashSet, LinkedHashSet, and TreeSet offer specific ways to implement these interactions. These classes use various data structures and algorithms to efficiently handle your data. With so many options available, the framework is super flexible and can handle all sorts of real-world data challenges.