# STACK

The `Stack` class in Java provides a simple implementation of a stack data structure. It is a subclass of `java.util.Vector` and offers various methods for managing a stack. Here are some of the most commonly used methods, categorized by their respective functionalities:

## PUSHING ELEMENTS ONTO THE STACK:

1. `push(E item)`: Pushes an element onto the stack.

## POPPING ELEMENTS FROM THE STACK:

1. `pop()`: Removes and returns the element at the top of the stack.

## PEEKING AT THE TOP ELEMENT:

1. `peek()`: Returns the element at the top of the stack without removing it.

## CHECKING IF THE STACK IS EMPTY:

1. `empty()`: Returns `true` if the stack is empty; otherwise, returns `false`.

## SEARCHING FOR AN ELEMENT:

1. `search(Object o)`: Searches for the specified element and returns its 1-based position from the top of the stack. Returns -1 if the element is not found.

## SIZE OF THE STACK:

1. `size()`: Returns the number of elements in the stack.

## ITERATING OVER THE STACK:

- While the `Stack` class does not provide dedicated iteration methods, you can use a loop or iterator to traverse the elements. Keep in mind that `Stack` is based on a `Vector` and is somewhat outdated. Consider using a `Deque` like `ArrayDeque` for stack-like behavior, which is part of the Java Collections Framework.

Here's an example of how to use some of these methods in a Java program:

```
import java.util.Stack;

public class StackExample {
   public static void main(String[] args) {
      Stack<Integer> stack = new Stack<>();

      stack.push(1);
      stack.push(2);
      stack.push(3);

      System.out.println("Top element: " + stack.peek()); // Prints 3
      System.out.println("Popped element: " + stack.pop()); // Prints 3
      System.out.println("Is the stack empty? " + stack.empty()); // Prints false
      System.out.println("Stack size: " + stack.size()); // Prints 2
      System.out.println("Position of 2 in the stack: " + stack.search(2)); // Prints 2
   }
}
```

This example demonstrates basic stack operations using the `Stack` class in Java.

# QUEUE

The `Queue` interface in Java represents a collection designed for holding elements before processing. Queues follow the "first-in, first-out" (FIFO) principle, where elements are processed in the order they were added. The `Queue` interface defines several methods for working with queues. Below are some of the commonly used methods, categorized by their functionalities:

## ADDING ELEMENTS TO THE QUEUE:

1. `add(E e)`: Adds the specified element to the queue. Throws an exception if the queue is full.

## OFFERING ELEMENTS TO THE QUEUE:

1. `offer(E e)`: Adds the specified element to the queue. Returns `true` if the element was added successfully, `false` if the queue is full.

## RETRIEVING AND REMOVING THE HEAD OF THE QUEUE:

1. `remove()`: Removes and returns the head of the queue. Throws an exception if the queue is empty.
2. `poll()`: Removes and returns the head of the queue. Returns `null` if the queue is empty.

## RETRIEVING, BUT NOT REMOVING THE HEAD OF THE QUEUE:

1. `element()`: Returns the head of the queue without removing it. Throws an exception if the queue is empty.
2. `peek()`: Returns the head of the queue without removing it. Returns `null` if the queue is empty.

## CHECKING IF THE QUEUE IS EMPTY:

1. `isEmpty()`: Returns `true` if the queue is empty; otherwise, returns `false`.

## SIZE OF THE QUEUE:

1. `size()`: Returns the number of elements in the queue.

## ITERATING OVER THE QUEUE:

- While the `Queue` interface does not define dedicated iteration methods, you can use an iterator to traverse the elements of a concrete implementation of the `Queue` interface, such as `LinkedList` or `ArrayDeque`.

Here's an example of how to use some of these methods with a `LinkedList`:

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        queue.offer("Apple");
        queue.offer("Banana");
        queue.offer("Cherry");

        System.out.println("Head of the queue: " + queue.peek()); // Prints "Apple"
        System.out.println("Removed element: " + queue.poll()); // Removes and prints "Apple"
        System.out.println("Is the queue empty? " + queue.isEmpty()); // Prints false
        System.out.println("Queue size: " + queue.size()); // Prints 2
    }
}
```

This example demonstrates basic queue operations using the `Queue` interface in Java, implemented with a `LinkedList`.

# LINKED LIST

The `LinkedList` class in Java, which is part of the Java Collections Framework, provides several methods for working with linked lists. Below is a list of some of the most commonly used methods, grouped by their respective functionalities:

ADDING ELEMENTS:

1. `add(E e)`: Appends the specified element to the end of the list.
2. `add(int index, E element)`: Inserts the specified element at the specified position.
3. `addFirst(E e)`: Inserts the specified element at the beginning of the list.
4. `addLast(E e)`: Appends the specified element to the end of the list.

REMOVING ELEMENTS:

1. `remove()`: Removes and returns the first element in the list.
2. `remove(int index)`: Removes and returns the element at the specified index.
3. `removeFirst()`: Removes and returns the first element in the list.
4. `removeLast()`: Removes and returns the last element in the list.
5. `remove(Object o)`: Removes the first occurrence of the specified element.
6. `clear()`: Removes all elements from the list.

ACCESSING ELEMENTS:

1. `get(int index)`: Returns the element at the specified index.
2. `getFirst()`: Returns the first element in the list.
3. `getLast()`: Returns the last element in the list.

CHECKING FOR ELEMENTS:

1. `contains(Object o)`: Checks if the list contains the specified element.
2. `isEmpty()`: Returns true if the list is empty.

ITERATING OVER ELEMENTS:

1. `iterator()`: Returns an iterator to traverse the elements in the list.
2. `listIterator()`: Returns a list iterator to traverse the elements bidirectionally.

SIZE AND INFORMATION:

1. `size()`: Returns the number of elements in the list.
2. `toArray()`: Converts the list to an array.

OTHER OPERATIONS:

1. `set(int index, E element)`: Replaces the element at the specified index with the specified element.
2. `indexOf(Object o)`: Returns the index of the first occurrence of the specified element.
3. `lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified element.

These are some of the commonly used methods provided by the `LinkedList` class in Java. You can use these methods to manipulate and work with linked lists in your Java programs.

```
import java.util.LinkedList;

public class JavaLinkedListExample {
   public static void main(String[] args) {
        LinkedList<Integer> linkedList = new LinkedList<>();

        linkedList.add(1);
        linkedList.add(2);
        linkedList.add(3);
        linkedList.add(4);

        System.out.println("Linked List: " + linkedList);

        linkedList.addFirst(0);
        linkedList.addLast(5);

        System.out.println("Updated Linked List: " + linkedList);

        linkedList.remove(Integer.valueOf(3));
        System.out.println("Linked List after removing '3': " + linkedList);

        int firstElement = linkedList.getFirst();
        int lastElement = linkedList.getLast();
        System.out.println("First Element: " + firstElement);
        System.out.println("Last Element: " + lastElement);
    }
}
```

In this example, we use the java.util.LinkedList class to create a linked list of integers. We perform operations such as adding elements, adding elements at the beginning and end, removing an element by value, and accessing the first and last elements. Finally, we print the contents of the linked list using System.out.println.

## ARRAY LIST

The `ArrayList` class in Java is part of the Java Collections Framework and provides a dynamic array implementation, allowing you to store and manage a list of elements. Here are some of the commonly used methods and functionalities of the `ArrayList` class:

ADDING ELEMENTS:

1. `add(E element)`: Appends the specified element to the end of the list.
2. `add(int index, E element)`: Inserts the specified element at the specified position.
3. `addAll(Collection<? extends E> c)`: Appends all elements from the specified collection to the end of the list.
4. `addAll(int index, Collection<? extends E> c)`: Inserts all elements from the specified collection at the specified position.

REMOVING ELEMENTS:

1. `remove(int index)`: Removes and returns the element at the specified index.
2. `remove(Object o)`: Removes the first occurrence of the specified element.
3. `clear()`: Removes all elements from the list.

ACCESSING ELEMENTS:

1. `get(int index)`: Returns the element at the specified index.
2. `set(int index, E element)`: Replaces the element at the specified index with the specified element.

CHECKING FOR ELEMENTS:

1. `contains(Object o)`: Checks if the list contains the specified element.

2. \`isEmpty()\`: Returns true if the list is empty.

SIZE AND CAPACITY:

1. \`size()\`: Returns the number of elements in the list.
2. \`ensureCapacity(int minCapacity)\`: Increases the capacity of the list to at least the specified minimum capacity.
3. \`trimToSize()\`: Reduces the capacity of the list to the current size.

ITERATING OVER ELEMENTS:

1. \`iterator()\`: Returns an iterator to traverse the elements in the list.
2. \`listIterator()\`: Returns a list iterator to traverse the elements bidirectionally.

SUBLIST:

1. \`subList(int fromIndex, int toIndex)\`: Returns a view of the list that contains elements in a specified range.

Here's an example of how to use some of these methods with an \`ArrayList\`:

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>();

        // Adding elements to the list
        arrayList.add("Alice");
        arrayList.add("Bob");
        arrayList.add("Charlie");

        // Getting an element by index
        String name = arrayList.get(1); // Retrieves "Bob"

        // Checking for element existence
        boolean containsCharlie = arrayList.contains("Charlie"); // Returns true

        // Removing an element
        arrayList.remove(2); // Removes "Charlie"

        // Size of the list
        int size = arrayList.size(); // Returns 2
    }
}
```

This example illustrates basic \`ArrayList\` operations in Java, including adding elements, getting elements, checking for element existence, removing elements, and more.

## HASHMAP

The \`HashMap\` class in Java is a part of the Java Collections Framework and is a popular data structure for implementing a key-value store. It is used to store and manage key-value pairs, where each key is associated with a value. Below are some of the commonly used methods and functionalities of the `HashMap` class:

ADDING KEY-VALUE PAIRS:

1. `put(K key, V value)`: Associates the specified value with the specified key in the map. If the key already exists, the previous value is replaced.

GETTING VALUES FROM THE HASHMAP:

1. `get(Object key)`: Returns the value associated with the specified key, or `null` if the key is not found in the map.

CHECKING FOR KEY EXISTENCE:

1. `containsKey(Object key)`: Checks if the map contains the specified key.
2. `containsValue(Object value)`: Checks if the map contains the specified value.

REMOVING KEY-VALUE PAIRS:

1. `remove(Object key)`: Removes the key and its associated value from the map.

SIZE AND EMPTY CHECK:

1. `size()`: Returns the number of key-value pairs in the map.
2. `isEmpty()`: Returns `true` if the map is empty; otherwise, returns `false`.

ITERATING OVER KEY-VALUE PAIRS:

1. `keySet()`: Returns a set of all keys in the map.
2. `values()`: Returns a collection of all values in the map.
3. `entrySet()`: Returns a set of key-value pairs as `Map.Entry` objects.

CLEARING THE HASHMAP:

1. `clear()`: Removes all key-value pairs from the map.

Here's an example of how to use some of these methods with a `HashMap`:

```
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> hashMap = new HashMap<>();

        // Adding key-value pairs
        hashMap.put("Alice", 25);
        hashMap.put("Bob", 30);
        hashMap.put("Charlie", 28);

        // Getting a value
        int age = hashMap.get("Alice"); // Retrieves 25

        // Checking for key existence
        boolean containsBob = hashMap.containsKey("Bob"); // Returns true

        // Removing a key-value pair
        hashMap.remove("Charlie");

        // Size and empty check
        int size = hashMap.size(); // Returns 2
        boolean isEmpty = hashMap.isEmpty(); // Returns false
    }
}
```

This example demonstrates basic `HashMap` operations in Java, including adding key-value pairs, getting values, checking for key existence, removing key-value pairs, and more.

## HASHTABLE

The `Hashtable` class in Java, part of the Java Collections Framework, is a synchronized and thread-safe data structure for implementing a key-value store. It is used to store and manage key-value pairs, where each key is associated with a value. Below are some of the commonly used methods and functionalities of the `Hashtable` class:

### ADDING KEY-VALUE PAIRS:

1. `put(K key, V value)`: Associates the specified value with the specified key in the map. If the key already exists, the previous value is replaced.

### GETTING VALUES FROM THE HASHTABLE:

1. `get(Object key)`: Returns the value associated with the specified key, or `null` if the key is not found in the map.

### CHECKING FOR KEY EXISTENCE:

1. `containsKey(Object key)`: Checks if the map contains the specified key.
2. `contains(Object value)`: Checks if the map contains the specified value.

### REMOVING KEY-VALUE PAIRS:

1. `remove(Object key)`: Removes the key and its associated value from the map.

### SIZE AND EMPTY CHECK:

1. `size()`: Returns the number of key-value pairs in the map.
2. `isEmpty()`: Returns `true` if the map is empty; otherwise, returns `false.

### ITERATING OVER KEY-VALUE PAIRS:

1. `keys()`: Returns an enumeration of all keys in the map.
2. `elements()`: Returns an enumeration of all values in the map.

### CLEARING THE HASHTABLE:

1. `clear()`: Removes all key-value pairs from the map.

Here's an example of how to use some of these methods with a `Hashtable`:

```
import java.util.Hashtable;
import java.util.Enumeration;

public class HashtableExample {
    public static void main(String[] args) {
        Hashtable<String, Integer> hashtable = new Hashtable<>();

        // Adding key-value pairs
        hashtable.put("Alice", 25);
        hashtable.put("Bob", 30);
        hashtable.put("Charlie", 28);

        // Getting a value
        int age = hashtable.get("Alice"); // Retrieves 25

        // Checking for key existence
        boolean containsBob = hashtable.containsKey("Bob"); // Returns true

        // Removing a key-value pair
        hashtable.remove("Charlie");

        // Size and empty check
        int size = hashtable.size(); // Returns 2
        boolean isEmpty = hashtable.isEmpty(); // Returns false

        // Iterating through keys and values
        Enumeration<String> keys = hashtable.keys();
        while (keys.hasMoreElements()) {
            String key = keys.nextElement();
            int value = hashtable.get(key);
            System.out.println(key + ": " + value);
        }
    }
}
```

This example demonstrates basic `Hashtable` operations in Java, including adding key-value pairs, getting values, checking for key existence, removing key-value pairs, iterating through keys and values, and more.

## HASHSET

The `HashSet` class in Java is part of the Java Collections Framework and provides an implementation of a set using a hash table. A `HashSet` is an unordered collection of unique elements. Here are some of the commonly used methods and functionalities of the `HashSet` class:

### ADDING ELEMENTS TO THE HASHSET:

1. `add(E e)`: Adds the specified element to the set if it is not already present. If the element is already in the set, the set remains unchanged.

### CHECKING FOR ELEMENT EXISTENCE:

1. `contains(Object o)`: Checks if the set contains the specified element.
2. `isEmpty()`: Returns `true` if the set is empty; otherwise, returns `false`.

### REMOVING ELEMENTS FROM THE HASHSET:

1. `remove(Object o)`: Removes the specified element from the set, if it is present.

### SIZE AND INFORMATION:

1. `size()`: Returns the number of elements in the set.
2. `toArray()`: Converts the set to an array.

### CLEARING THE HASHSET:

1. `clear()`: Removes all elements from the set.

Here's an example of how to use some of these methods with a `HashSet`:

```
import java.util.HashSet;

public class HashSetExample {
  public static void main(String[] args) {
    HashSet<String> hashSet = new HashSet<>();

    // Adding elements to the set
    hashSet.add("Apple");
    hashSet.add("Banana");
    hashSet.add("Cherry");

    // Checking for element existence
    boolean containsBanana = hashSet.contains("Banana"); // Returns true

    // Removing an element
    hashSet.remove("Cherry");

    // Size and empty check
    int size = hashSet.size(); // Returns 2
    boolean isEmpty = hashSet.isEmpty(); // Returns false
  }
}
```

This example demonstrates basic `HashSet` operations in Java, including adding elements, checking for element existence, removing elements, and more.

## TREESET

The `TreeSet` class in Java, part of the Java Collections Framework, provides an implementation of a set based on a Red-Black tree. A `TreeSet` is an ordered collection of unique elements. Here are some of the commonly used methods and functionalities of the `TreeSet` class:

### ADDING ELEMENTS TO THE TREESET:

1. `add(E e)`: Adds the specified element to the set if it is not already present. If the element is already in the set, the set remains unchanged.

### CHECKING FOR ELEMENT EXISTENCE:

1. `contains(Object o)`: Checks if the set contains the specified element.
2. `isEmpty()`: Returns `true` if the set is empty; otherwise, returns `false`.

### REMOVING ELEMENTS FROM THE TREESET:

1. `remove(Object o)`: Removes the specified element from the set, if it is present.

### GETTING ELEMENTS FROM THE TREESET:

1. `first()`: Returns the first (lowest) element in the set.
2. `last()`: Returns the last (highest) element in the set.
3. `ceiling(E e)`: Returns the least element in the set greater than or equal to the given element.
4. `floor(E e)`: Returns the greatest element in the set less than or equal to the given element.

SIZE AND INFORMATION:

1. `size()`: Returns the number of elements in the set.

ITERATING OVER ELEMENTS:

1. `iterator()`: Returns an iterator to traverse the elements in ascending order.
2. `descendingIterator()`: Returns an iterator to traverse the elements in descending order.

SUBSETS:

1. `subSet(E fromElement, E toElement)`: Returns a view of the set that contains elements in a specified range.
2. `headSet(E toElement)`: Returns a view of the set that contains elements less than the specified element.
3. `tailSet(E fromElement)`: Returns a view of the set that contains elements greater than or equal to the specified element.

CLEARING THE TREESET:

1. `clear()`: Removes all elements from the set.

Here's an example of how to use some of these methods with a `TreeSet`:

```
import java.util.TreeSet;

public class TreeSetExample {
   public static void main(String[] args) {
      TreeSet<Integer> treeSet = new TreeSet<>();

      // Adding elements to the set
      treeSet.add(3);
      treeSet.add(1);
      treeSet.add(2);

      // Checking for element existence
      boolean containsTwo = treeSet.contains(2); // Returns true

      // Removing an element
      treeSet.remove(1);

      // Getting the first and last elements
      int firstElement = treeSet.first(); // Returns 2
      int lastElement = treeSet.last(); // Returns 3

      // Size of the set
      int size = treeSet.size(); // Returns 2
   }
}
```

This example illustrates basic `TreeSet` operations in Java, including adding elements, checking for element existence, removing elements, and more.

# TREEMAP

The `TreeMap` class in Java, part of the Java Collections Framework, provides an implementation of a sorted map based on a Red-Black tree. A `TreeMap` is an ordered collection of key-value pairs where the keys are maintained in sorted order. Here are some of the commonly used methods and functionalities of the `TreeMap` class:

ADDING KEY-VALUE PAIRS TO THE TREEMAP:

1. `put(K key, V value)`: Associates the specified value with the specified key in the map. If the key already exists, the previous value is replaced.

GETTING VALUES FROM THE TREEMAP:

1. `get(Object key)`: Returns the value associated with the specified key, or `null` if the key is not found in the map.

CHECKING FOR KEY EXISTENCE:

1. `containsKey(Object key)`: Checks if the map contains the specified key.
2. `containsValue(Object value)`: Checks if the map contains the specified value.

REMOVING KEY-VALUE PAIRS FROM THE TREEMAP:

1. `remove(Object key)`: Removes the key and its associated value from the map.

GETTING INFORMATION ABOUT THE TREEMAP:

1. `size()`: Returns the number of key-value pairs in the map.
2. `isEmpty()`: Returns `true` if the map is empty; otherwise, returns `false`.
3. `firstKey()`: Returns the first (lowest) key in the map.
4. `lastKey()`: Returns the last (highest) key in the map.
5. `ceilingKey(K key)`: Returns the least key greater than or equal to the given key.
6. `floorKey(K key)`: Returns the greatest key less than or equal to the given key.

ITERATING OVER KEY-VALUE PAIRS:

1. `keySet()`: Returns a set of all keys in the map.
2. `values()`: Returns a collection of all values in the map.
3. `entrySet()`: Returns a set of key-value pairs as `Map.Entry` objects.

SUBMAPS:

1. `subMap(K fromKey, K toKey)`: Returns a view of the map containing key-value pairs in a specified range.
2. `headMap(K toKey)`: Returns a view of the map containing key-value pairs with keys less than the specified key.
3. `tailMap(K fromKey)`: Returns a view of the map containing key-value pairs with keys greater than or equal to the specified key.

CLEARING THE TREEMAP:

1. `clear()`: Removes all key-value pairs from the map.

Here's an example of how to use some of these methods with a `TreeMap`:

```
import java.util.TreeMap;

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

    // Adding key-value pairs to the map
    treeMap.put("Alice", 25);
    treeMap.put("Bob", 30);
    treeMap.put("Charlie", 28);
```

```
        // Getting a value using a key
        int age = treeMap.get("Alice"); // Retrieves 25

        // Checking for key existence
        boolean containsBob = treeMap.containsKey("Bob"); // Returns true

        // Removing a key-value pair
        treeMap.remove("Charlie");

        // Size and empty check
        int size = treeMap.size(); // Returns 2
        boolean isEmpty = treeMap.isEmpty(); // Returns false
    }
}
```

This example demonstrates basic `TreeMap` operations in Java, including adding key-value pairs, getting values, checking for key existence, removing key-value pairs, and more.