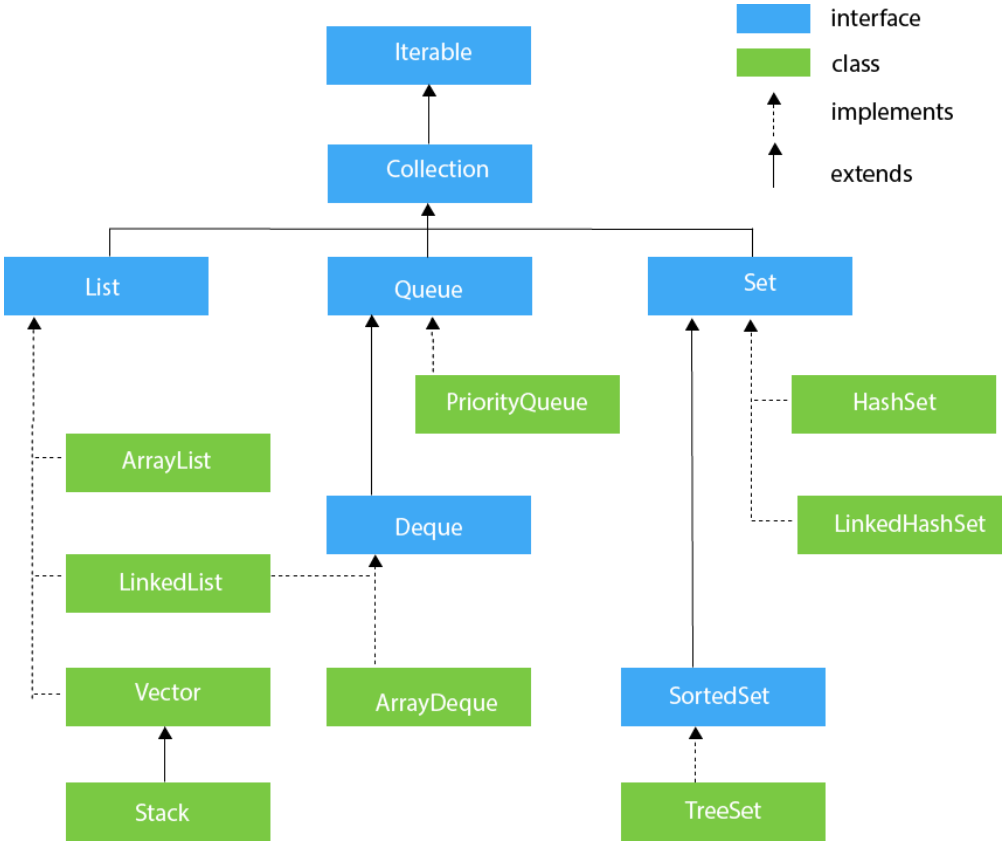
1. Collection Framework
   - The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
   - Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
   - What is Collection framework
     - The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:
       - Interfaces and its implementations, i.e., classes
       - Algorithm
     - Hierarchy of Collection Framework:
       Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces for the Collection framework.



     - Iterable Interface
       - The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
     - Collection Interface
       - The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection

interface builds the foundation on which the collection framework depends.

- List Interface
  - List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.
  - List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
- Queue Interface
  - Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
- Set Interface
  - Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

2. Arrays class

- The Arrays class in java.util package is a part of the Java Collection Framework. This class provides static methods to dynamically create and access Java arrays.

Ex.

```
// Java Program to Demonstrate Arrays Class
// Via binarySearch() method

// Importing Arrays utility class
// from java.util package
import java.util.Arrays;

// Main class
public class Arrays1 {

  // Main driver method
  public static void main(String[] args)
  {

    // Get the Array
    int intArr[] = { 10, 20, 15, 22, 35 };

    Arrays.sort(intArr);

    int intKey = 22;

    // Print the key and corresponding index
```

```
        System.out.println(
            intKey + " found at index = "
            + Arrays.binarySearch(intArr, intKey));
    }
}
```

Output: 22 found at index = 3

3. ArrayList class
    - The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types.
    - The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

```
Ex.
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:
Ravi
Vijay
Ravi
Ajay

4. LinkedList class
    - LinkedList implements the Collection interface.
    - It uses a doubly linked list internally to store the elements.
    - It can store the duplicate elements.
    - It maintains the insertion order and is not synchronized.
    - In LinkedList, the manipulation is fast because no shifting is required.

Ex.

```
import java.util.LinkedList;
public class TestJavaCollection2
{
public static void main(String args[])
{
LinkedList<String> al=new LinkedList<String>();
al.add("Needa");
al.add("Hardik");
al.add("Jay");
al.add("Raj");

System.out.println("List:"+al);

 al.add(1,"Ranjeet");

 System.out.println("Updated list:");
 System.out.println("List:"+al);

 String str = al.get(1);
System.out.println("str:"+str);

al.set(1, "Kajal");
System.out.println("Updated list:"+ al);

str = al.remove(1);
System.out.println("Removed member:"+str);

System.out.println("Linked list contains:"+al.contains("Needa"));
System.out.println("Linked list contains:"+al.contains("Kajal"));

int i = al.indexOf("Needa");
System.out.println("index of:"+ "Needa:"+i);

if(al.contains("Needa"))
{
   i = al.indexOf("Needa");
   al.set(i,"Hirva");

   System.out.printf("Updated:"+al);
}
```

```
}
}
```

Output:

```
List:[Needa, Hardik, Jay, Raj]
Updated list:
List:[Needa, Ranjeet, Hardik, Jay, Raj]
str:Ranjeet
Updated list:[Needa, Kajal, Hardik, Jay, Raj]
Removed member:Kajal
Linked list contains:true
Linked list contains:false
index of:Needa:0
Updated:[Hirva, Hardik, Jay, Raj]
```

5. ListIterator interface
   - The ListIterator interface of the Java collections framework provides the functionality to access elements of a list.
   - It is bidirectional. This means it allows us to iterate elements of a list in both the direction.
   - It extends the Iterator interface.
   - Methods of ListIterator
     - hasNext() - returns true if there exists an element in the list
     - next() - returns the next element of the list
     - nextIndex() returns the index of the element that the next() method will return
     - previous() - returns the previous element of the list
     - previousIndex() - returns the index of the element that the previous() method will return
     - remove() - removes the element returned by either next() or previous()
     - set() - replaces the element returned by either next() or previous() with the specified element
   - Ex.

```
import java.util.ArrayList;
import java.util.ListIterator;

class TestJavaCollection3
{
  public static void main(String[] args)
  {
    // Creating an ArrayList
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(3);
    numbers.add(2);
    System.out.println("ArrayList: " + numbers);

    // Creating an instance of ListIterator
```

```
        ListIterator<Integer> iterate = numbers.listIterator();

        // Using the next() method
        int number1 = iterate.next();
        System.out.println("Next Element: " + number1);

        // Using the nextIndex()
        int index1 = iterate.nextIndex();
        System.out.println("Position of Next Element: " + index1);

        // Using the hasNext() method
        System.out.println("Is there any next element? " + iterate.hasNext());

        iterate = numbers.listIterator();
        iterate.next();
        iterate.next();

        // Using the previous() method
         number1 = iterate.previous();
        System.out.println("Previous Element: " + number1);

        // Using the previousIndex()
         index1 = iterate.previousIndex();
        System.out.println("Position of the Previous element: " + index1);

    }
}
```

Output:
ArrayList: [1, 3, 2]
Next Element: 1
Position of Next Element: 1
Is there any next element? true
Previous Element: 3
Position of the Previous element: 0

6. HashSet class
   - Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
   - HashSet stores the elements by using a mechanism called hashing.
   - HashSet contains unique elements only.
   - HashSet allows null value.
   - HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
   - HashSet is the best approach for search operations.

- Ex.

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
  HashSet<String> set=new HashSet<String>();
  set.add("Manthan");
  set.add("Priyanshi");
  set.add("Rushi");
  set.add("Titiksha");
  set.add("Manthan");
  //Traversing elements
   System.out.println("Hash set:"+set);

 }
}
```

Output:

```
Hash set:[Manthan, Titiksha, Priyanshi, Rushi]
```

Ex.

```
import java.util.*;
class HashSet2{
 public static void main(String args[]){
  HashSet<String> set=new HashSet<String>();
        set.add("Ambika");
        set.add("Chaitnya");
        set.add("Jimeet");
        set.add("Kushal");
        System.out.println("An initial list of elements: "+set);
        //Removing specific element from HashSet
        set.remove("Jimeet");
        System.out.println("After invoking remove(object) method: "+set);
        HashSet<String> set1=new HashSet<String>();
        set1.add("Karan");
        set1.add("Madhu");
        set.addAll(set1);
        System.out.println("Updated List: "+set);
        //Removing all the new elements from HashSet
        set.removeAll(set1);
        System.out.println("After invoking removeAll() method: "+set);
        //Removing elements on the basis of specified condition
        set.removeIf(str->str.contains("Kushal"));
        System.out.println("After invoking removeIf() method: "+set);
        //Removing all the elements available in the set
        set.clear();
```

```
        System.out.println("After invoking clear() method: "+set);
 }
}
```

Output:

```
An initial list of elements: [Ambika, Chaitnya, Jimeet, Kushal]
After invoking remove(object) method: [Ambika, Chaitnya, Kushal]
Updated List: [Ambika, Chaitnya, Karan, Madhu, Kushal]
After invoking removeAll() method: [Ambika, Chaitnya, Kushal]
After invoking removeIf() method: [Ambika, Chaitnya]
After invoking clear() method: []
```

Ex.

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
}
public class HashSetExample {
public static void main(String[] args) {
   HashSet<Book> set=new HashSet<Book>();
   //Creating Books
   Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
   Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw
Hill",4);
   Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
   //Adding Books to HashSet
   set.add(b1);
   set.add(b2);
   set.add(b3);
   //Traversing HashSet
   for(Book b:set){
   System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
   }
}
}
```

Output:
101 Let us C Yashwant Kanetkar BPB 8
103 Operating System Galvin Wiley 6
102 Data Communications & Networking Forouzan Mc Graw Hill 4

7. LinkedHashSet class
   - Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.
   - Java LinkedHashSet class contains unique elements only like HashSet.
   - Java LinkedHashSet class provides all optional set operations and permits null elements.
   - Java LinkedHashSet class is non-synchronized.
   - Java LinkedHashSet class maintains insertion order.

```
Ex.
import java.util.*;
class LinkedHashSet1{
 public static void main(String args[]){
  LinkedHashSet<String> al=new LinkedHashSet<String>();
  al.add("AB");
  al.add("AC");
  al.add("AB");
  al.add("AD");

  System.out.println("LinkedHashSet:"+al);

  // Removing element
  al.remove("AC");
  System.out.println(" After removing LinkedHashSet:"+al);

  LinkedHashSet<String> lhs=new LinkedHashSet<String>();
  lhs.add("Ajay");
  lhs.add("Vijay");

  // Add all
  al.addAll(lhs);
  System.out.println(" After adding LinkedHashSet:"+al);

 //Intersection of Sets

 al.clear();

 al.add("A");
 al.add("B");
 lhs.add("A");
 lhs.add("C");
```

```
al.retainAll(lhs);
System.out.println(" Intersection of Hashset"+al);

// Difference of Sets
al.clear();
al.add("A");
al.add("B");
lhs.add("A");
lhs.add("C");
al.removeAll(lhs);

System.out.println("Difference of Hashset"+al);


}
}
```

Output:

```
LinkedHashSet:[AB, AC, AD]
After removing LinkedHashSet:[AB, AD]
After adding LinkedHashSet:[AB, AD, Ajay, Vijay]
Intersection of Hashset[A]
Difference of Hashset[B]
```

8. TreeSet class
   - Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.
   - TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time.
   - Java TreeSet class contains unique elements only like HashSet.
   - Java TreeSet class access and retrieval times are quiet fast.
   - Java TreeSet class doesn't allow null element.
   - Java TreeSet class is non synchronized.
   - Java TreeSet class maintains ascending order.
   - The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

EX.

```
import java.util.TreeSet;

import javax.swing.plaf.synth.SynthCheckBoxMenuItemUI;
class TreeSet1 {
   public static void main(String[] args) {

      TreeSet<Integer> evenNumbers = new TreeSet<>();

      // Using the add() method
      evenNumbers.add(2);
      evenNumbers.add(4);
      evenNumbers.add(6);
      System.out.println("TreeSet: " + evenNumbers);

      TreeSet<Integer> numbers = new TreeSet<>();
      numbers.add(1);

      // Using the addAll() method
      numbers.addAll(evenNumbers);
      System.out.println("New TreeSet: " + numbers);

      //Remove Elements

      //Using the remove() method
      boolean value1 = numbers.remove(4);
      System.out.println("Is 4 removed? " + value1);

      // Using the removeAll() method
      boolean value2 = numbers.removeAll(numbers);
      System.out.println("Are all elements removed? " + value2);

      System.out.println("Tree set:"+ numbers);

      //Methods for Navigation
      numbers.add(2);
      numbers.add(5);
      numbers.add(6);

      System.out.println("TreeSet: " + numbers);
      // Using the first() method
      int first = numbers.first();
      System.out.println("First Number: " + first);

      // Using the last() method
```

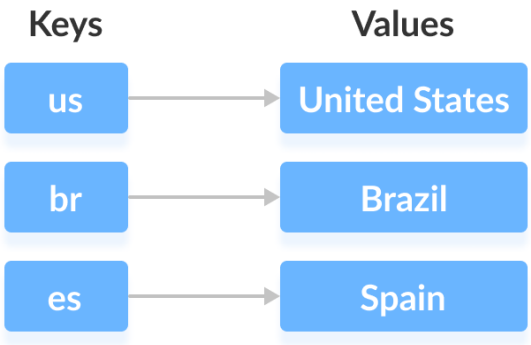
```
int last = numbers.last();
System.out.println("Last Number: " + last);

// ceiling(), floor(), higher() and lower() Methods
numbers.add(1);
numbers.add(3);
numbers.add(4);
numbers.add(7);
numbers.add(8);
numbers.add(9);

System.out.println("Tree set:"+ numbers);
// Using higher()
System.out.println("Using higher: " + numbers.higher(5));

// Using lower()
System.out.println("Using lower: " + numbers.lower(5));

// Using ceiling() Returns the lowest element among those elements that are greater than the specified element
//If the element passed exists in a tree set, it returns the element passed as an argument.
System.out.println("Using ceiling: " + numbers.ceiling(5));

// Using floor() Returns the greatest element among those elements that are less than the specified element
//If the element passed exists in a tree set, it returns the element passed as an argument.
System.out.println("Using floor: " + numbers.floor(5));

//pollfirst() and pollLast() Methods

//// Using pollFirst()
System.out.println("Removed First Element: " + numbers.pollFirst());

// Using pollLast()
System.out.println("Removed Last Element: " + numbers.pollLast());

System.out.println("New TreeSet: " + numbers);

//headSet(), tailSet() and subSet() Methods
// Using headSet() with default boolean value
System.out.println("Using headSet without boolean value: " + numbers.headSet(5));

// Using headSet() with specified boolean value
```

```
        System.out.println("Using headSet with boolean value: " + numbers.headSet(5,true));

        // Using tailSet() with default boolean value
        System.out.println("Using tailSet without boolean value: " + numbers.tailSet(5));

        // Using tailSet() with specified boolean value
        System.out.println("Using tailSet with boolean value: " + numbers.tailSet(5, false));

        // subSet(e1, bv1, e2, bv2)
         // Using subSet() with specified boolean value
         System.out.println("Using subSet with boolean value: " + numbers.subSet(4, false, 7, true));
      }
   }
```

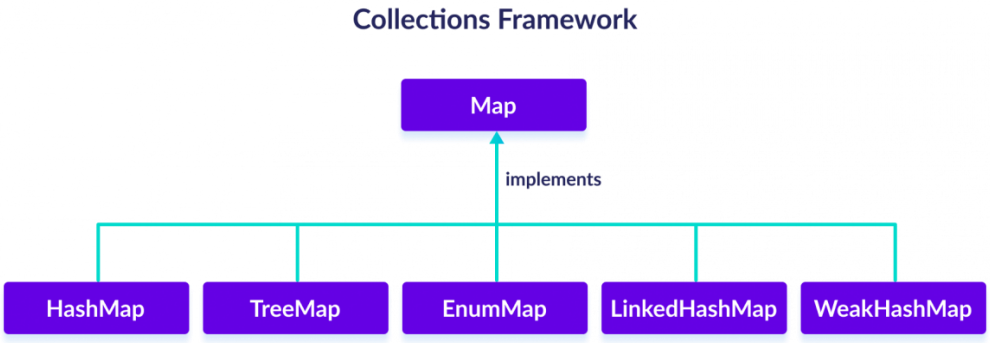
Output:

```
TreeSet: [2, 4, 6]
New TreeSet: [1, 2, 4, 6]
Is 4 removed? true
Are all elements removed? true
Tree set:[]
TreeSet: [2, 5, 6]
First Number: 2
Last Number: 6
Tree set:[1, 2, 3, 4, 5, 6, 7, 8, 9]
Using higher: 6
Using lower: 4
Using ceiling: 5
Using floor: 5
Removed First Element: 1
Removed Last Element: 9
New TreeSet: [2, 3, 4, 5, 6, 7, 8]
Using headSet without boolean value: [2, 3, 4]
Using headSet with boolean value: [2, 3, 4, 5]
Using tailSet without boolean value: [5, 6, 7, 8]
Using tailSet with boolean value: [6, 7, 8]
Using subSet with boolean value: [5, 6, 7]
```

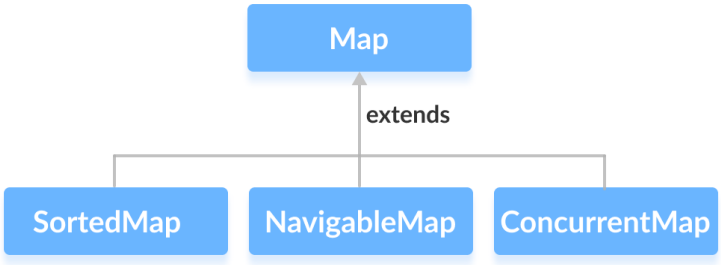
9. Map interface
   - The Map interface of the Java collections framework provides the functionality of the map data structure.
   - In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual Values.
   - A map cannot contain duplicate keys. And, each key is associated with a single value.



- We can access and modify values using the keys associated with them.
- In the above diagram, we have values: United States, Brazil, and Spain. And we have corresponding keys: us, br, and es.
- Since Map is an interface, we cannot create objects from it.
- In order to use functionalities of the Map interface, we can use these classes: HashMap, EnumMap, LinkedHashMap, WeakHashMap, TreeMap.



- The Map interface is also extended by these subinterfaces: SortedMap, NavigableMap, ConcurrentMap.



Ex.

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

class HashMap1 {

  public static void main(String[] args) {
    // Creating a map using the HashMap
    Map<String, Integer> numbers = new HashMap<>();

    // Insert elements to the map
    numbers.put("One", 1);
    numbers.put("Two", 2);
    System.out.println("Map: " + numbers);

    // Access keys of the map
    System.out.println("Keys: " + numbers.keySet());

    // Access values of the map
    System.out.println("Values: " + numbers.values());

    // Access entries of the map
    System.out.println("Entries: " + numbers.entrySet());

    // Remove Elements from the map
    int value = numbers.remove("Two");
    System.out.println("Removed Value: " + value);
  }
}
```

Output:

```
Map: {One=1, Two=2}
Keys: [One, Two]
Values: [1, 2]
Entries: [One=1, Two=2]
Removed Value: 2
```

10. TreeMap class
    - Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.
    - Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
    - Java TreeMap contains only unique elements.
    - Java TreeMap cannot have a null key but can have multiple null values.
    - Java TreeMap maintains ascending order.
    - Java TreeMap is non synchronized.

Ex.

```
import java.util.TreeMap;

class TreeMap1 {
   public static void main(String[] args) {
      // Creating TreeMap of even numbers
      TreeMap<String, Integer> evenNumbers = new TreeMap<>();

      // Using put()
      evenNumbers.put("Two", 2);
      evenNumbers.put("Four", 4);

      // Using putIfAbsent()
      evenNumbers.putIfAbsent("Six", 6);
      System.out.println("TreeMap of even numbers: " + evenNumbers);

      //Creating TreeMap of numbers
      TreeMap<String, Integer> numbers = new TreeMap<>();
      numbers.put("One", 1);

      // Using putAll()
      numbers.putAll(evenNumbers);
      System.out.println("TreeMap of numbers: " + numbers);

      // Access TreeMap Elements Using entrySet(), keySet() and values()
      // Using entrySet()
      System.out.println("Key/Value mappings: " + numbers.entrySet());

      // Using keySet()
      System.out.println("Keys: " + numbers.keySet());

      // Using values()
      System.out.println("Values: " + numbers.values());

      // Using get() and getOrDefault()

      // get() - Returns the value associated with the specified key. Returns null if the key
is not found.
      // Using get()
      numbers.put("Three", 3);

      int value1 = numbers.get("Three");
      System.out.println("Using get(): " + value1);
```

```
        //getOrDefault() - Returns the value associated with the specified key. Returns the specified default value if the key is not found.
        // Using getOrDefault()
        int value2 = numbers.getOrDefault("Five", 5);  // Try with key:"Three"
        System.out.println("Using getOrDefault(): " + value2);


        // remove method with single parameter
        int value = numbers.remove("Two");
        System.out.println("Removed value: " + value);

        // remove method with two parameters
        boolean result = numbers.remove("Three", 3);
        System.out.println("Is the entry {Three=3} removed? " + result);

        System.out.println("Updated TreeMap: " + numbers);

        // Methods for Navigation
        // Using the firstKey() method
        String firstKey = numbers.firstKey();
        System.out.println("First Key: " + firstKey);

        // Using the lastKey() method
        String lastKey = numbers.lastKey();
        System.out.println("Last Key: " + lastKey);

        // Using firstEntry() method
        System.out.println("First Entry: " + numbers.firstEntry());

        // Using the lastEntry() method
        System.out.println("Last Entry: " + numbers.lastEntry());

    }
}
```

Output:

```
TreeMap of even numbers: {Four=4, Six=6, Two=2}
TreeMap of numbers: {Four=4, One=1, Six=6, Two=2}
Key/Value mappings: [Four=4, One=1, Six=6, Two=2]
Keys: [Four, One, Six, Two]
Values: [4, 1, 6, 2]
Using get(): 3
Using getOrDefault(): 5
Removed value: 2
Is the entry {Three=3} removed? true
Updated TreeMap: {Four=4, One=1, Six=6}
```

```
First Key: Four
Last Key: Six
First Entry: Four=4
Last Entry: Six=6
```

11. StreamTokenizer class:
    - Java StreamTokenizer class to parse an input stream into tokens. We can use this class to break the InputStream object or an object of type Reader into tokens based on different identifiers, numbers, quoted strings, and various comment styles.
    - The use of InputStream as the input parameter is deprecated, so we will focus on using a Reader Object as input.

    Ex.

```
import java.io.IOException;
import java.io.Reader;
import java.io.StreamTokenizer;
import java.io.StringReader;

public class JavaStreamTokenizerExample {

        public static void main(String[] args) throws IOException {
                Reader reader = new StringReader("This is a test string for Stream Tokenizer Example");
                StreamTokenizer tokenizer = new StreamTokenizer(reader);
                while(tokenizer.nextToken()!=StreamTokenizer.TT_EOF){
                        System.out.println(tokenizer.sval);
                }

        }
}
```