JAVA Programming

TOPICs to be discussed

- Set Interface
  - ❑HashSet ❑LinkedHashSet
- SortedSet Interface
  - ❑TreeSet
- Map Interface
  - ❑HashMap ❑LinkedHashMap
- SortedMap Interface
  - ❑TreeMap

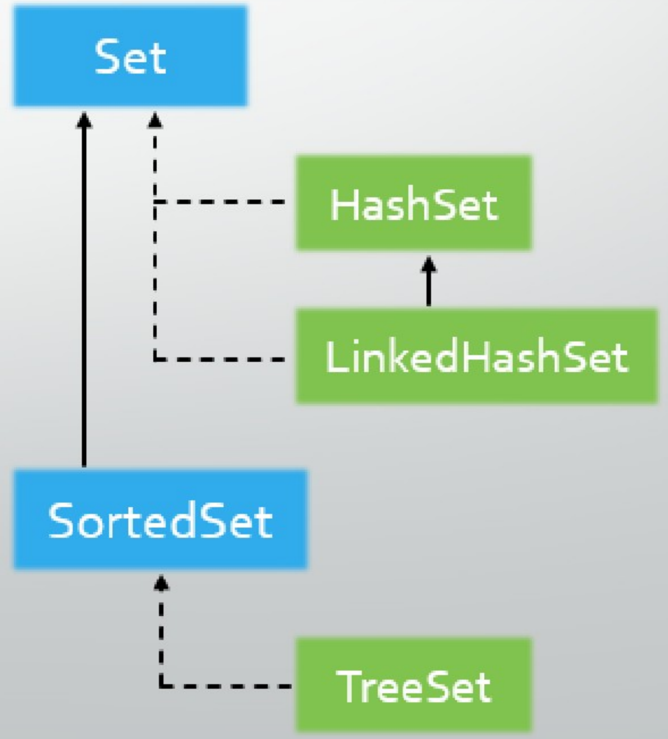
Let’s START …!!!

Set Interface

- The Set interface represents a collection of unique elements (no duplicates).
- It is part of the java.util package.
- It allows at most one **null element** (depending on the implementation).

Implementation

- ❑ HashSet
- ❑ LinkedHashSet
- ❑ TreeSet (implements SortedSet interface that extends Set Interface)



HashSet

- The HashSet class in **Java** is a part of the Java Collection Framework, implementing the Set interface.
- It is one of the most used implementations of the Set interface.

Key Features of HashSet

- Uses hashing for storage, making **add()**, **remove()**, and **contains()** operations O(1) on average.
- Does not maintain the order of elements.
- Allows a single **null element**.
- Not synchronized. If **thread safety** is needed, use Collections.synchronizedSet().

HashSet (Example)

```
import java.util.HashSet;
class HashSetDemo {
   public static void main(String[] args) {
       HashSet<String> hashSet = new HashSet<>();
       //Adding elements to the HashSet
       hashSet.add("Apple");  hashSet.add("Banana");
       hashSet.add("Cherry");
       hashSet.add("Apple");  //Duplicate, will not be added
       //Checking for an element
       System.out.println("Contains 'Banana': " +

hashSet.contains("Banana"));
       //Iterating through the HashSet
       for (String item : hashSet)
          System.out.println(item);
       //Removing an element
       hashSet.remove("Cherry");
       System.out.println("After removal: " + hashSet);
       //Checking size
       System.out.println("Size: " + hashSet.size());
   }
}
```

Output:

```
Contains 'Banana': true
Apple
Cherry
Banana
After removal: [Apple, Banana]
Size: 2
```

Note: The order of elements while printing can be anything, as it is not maintained in HashSets

LinkedHashSet

- The LinkedHashSet class in **Java** is a part of the java.util package.
- It extends HashSet and implements the Set interface, maintaining the insertion order of elements. This makes it different from HashSet, which does not guarantee any specific order.

Key Features of LinkedHashSet

- Internally, it uses a combination of a **hash table** (for fast operations like **add()**, **remove()**, **contains()**) and a **linked list** (to maintain the order).
- Allows a single **null element**.
- Like HashSet, it is not thread-synchronized. Requires external **synchronization** for concurrent access using Collections.synchronizedSet().

LinkedHashSet (Example)

```
import java.util.LinkedHashSet;
class LinkedHashSetDemo {
   public static void main(String[] args) {
       LinkedHashSet<String> linkedHashSet =

         new LinkedHashSet<>();
       //Adding elements to the HashSet
       linkedHashSet.add("Apple");
       linkedHashSet.add("Banana");
       linkedHashSet.add("Cherry");
       linkedHashSet.add("Apple");  //Duplicate, will not be added
       //Checking for an element
       System.out.println("Contains 'Banana': " +

linkedHashSet.contains("Banana"));
       //Iterating through the HashSet
       for (String item : linkedHashSet)
          System.out.println(item);
       //Removing an element
       linkedHashSet.remove("Cherry");
       System.out.println("After removal: " + linkedHashSet);
       //Checking size
       System.out.println("Size: " + linkedHashSet.size());
   }
}
```

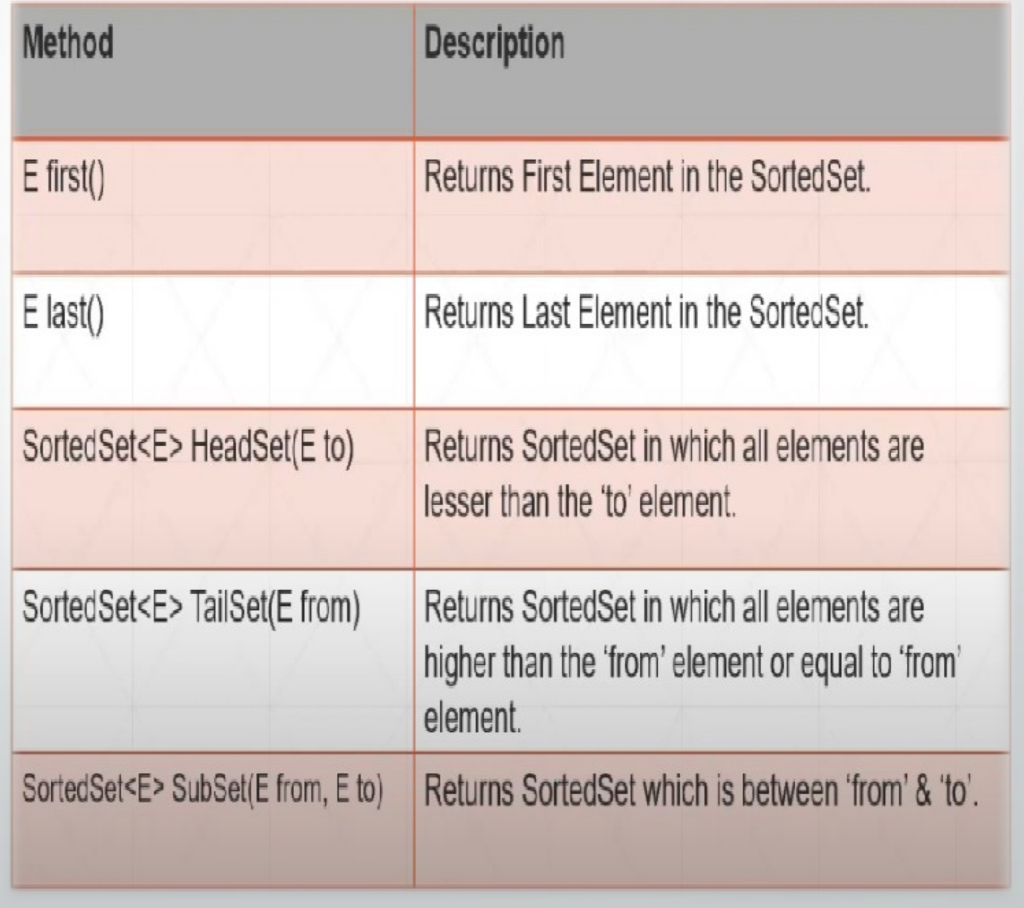
Output:

```
Contains 'Banana': true
Apple
Banana
Cherry
After removal: [Apple, Banana]
Size: 2
```

Note: The order of elements while printing is maintained in LinkedHashSets as the order of insertion of the elements

SortedSet Interface

- The SortedSet interface is part of the java.util package in **Java**.
- It extends the Set interface and represents a collection of unique elements that are maintained in sorted order.
- Sorting can be based on the natural order of the elements (if they implement Comparable) or a custom order defined by a Comparator.



Method	Description
E first()	Returns First Element in the SortedSet.
E last()	Returns Last Element in the SortedSet.
SortedSet<E> HeadSet(E to)	Returns SortedSet in which all elements are lesser than the 'to' element.
SortedSet<E> TailSet(E from)	Returns SortedSet in which all elements are higher than the 'from' element or equal to 'from' element.
SortedSet<E> SubSet(E from, E to)	Returns SortedSet which is between 'from' & 'to'.

Implementation

- ❑ TreeSet
- ❑ ConcurrentSkipListSet

TreeSet

- The TreeSet class in **Java** is part of the java.util package and implements the SortedSet and NavigableSet interfaces.
- It is a collection that maintains unique elements in sorted order. Internally, it uses a **red-black tree**, which is a self-balancing **binary search tree**, to store elements.

Key Features of TreeSet

- Operations like **add()**, **remove()**, and **contains()** take **O(log n)** time due to the underlying **red-black tree.**
- Elements are stored in ascending order by default. A custom Comparator can be used for user-defined sorting. (e.g., Comparator.reverseOrder() in param. constructor)
- does not allow **null elements** (else would have thrown NullPointerException).
- Not synchronized. For **thread safety** (if needed), use Collections.synchronizedSet().
- Provides methods for navigation like **higher()**, **lower()**, **ceiling()**, **floor()**, and range views like **subSet()**, **headSet()**, and **tailSet()**.

TreeSet (Example)

```java
import java.util.TreeSet;
class TreeSetDemo {
   public static void main(String[] args) {
      //Creating a TreeSet
      TreeSet<Integer> treeSet = new TreeSet<>();
      //Adding elements
      treeSet.add(50);    treeSet.add(20);
      treeSet.add(30);    treeSet.add(10);
      //Displaying elements (sorted order)
      System.out.println("TreeSet: " + treeSet);
      //Accessing first and last elements
      System.out.println("First: " + treeSet.first());
      System.out.println("Last: " + treeSet.last());
      //Subsets
      System.out.println("less than 30: " + treeSet.headSet(30));
      System.out.println("30 or greater: " + treeSet.tailSet(30));
      System.out.println("20 to 50: " + treeSet.subSet(20, 50));
      //Navigable methods
      System.out.println("Ceiling of 20: " + treeSet.ceiling(20));
      System.out.println("Floor of 20: " + treeSet.floor(20));
      System.out.println("Higher than 20: " + treeSet.higher(20));
      System.out.println("Lower than 20: " + treeSet.lower(20));
   }
}
```

Output:

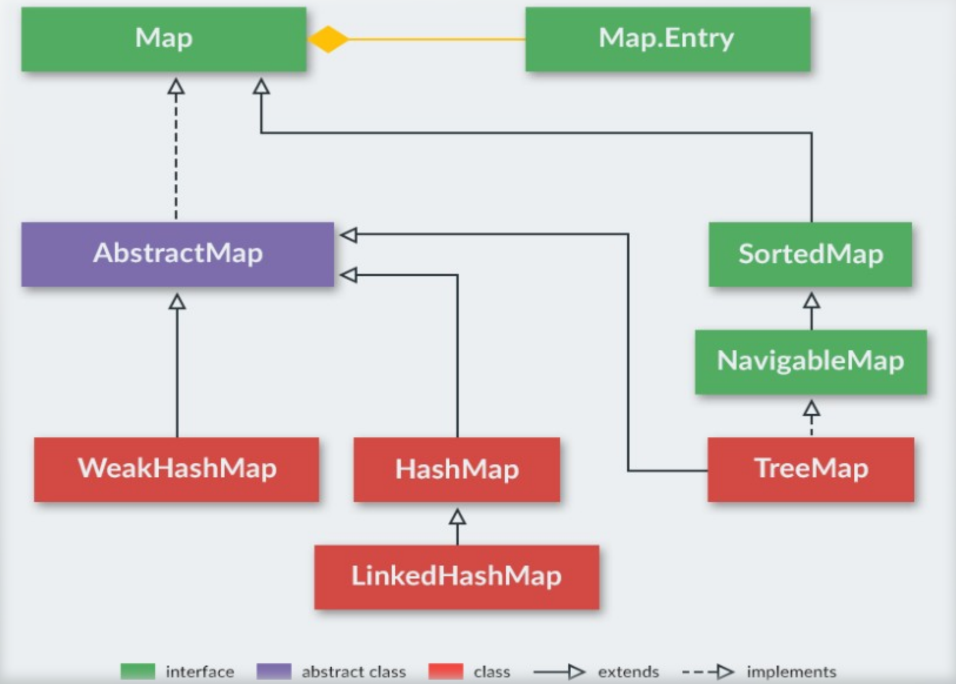
```
TreeSet: [10, 20, 30, 50]
First: 10
Last: 50
less than 30: [10, 20]
30 or greater: [30, 50]
20 to 50: [20, 30]
Ceiling of 20: 20
Floor of 20: 20
Higher than 20: 30
Lower than 20: 10
```

Map Interface

- The Map interface in **Java**, part of the java.util package, represents a collection of key-value pairs.
- It is designed for scenarios where data is stored and accessed using unique keys.
- Unlike other collection interfaces such as **List** or **Set**, a **Map** is not a subtype of Collection.

Implementation

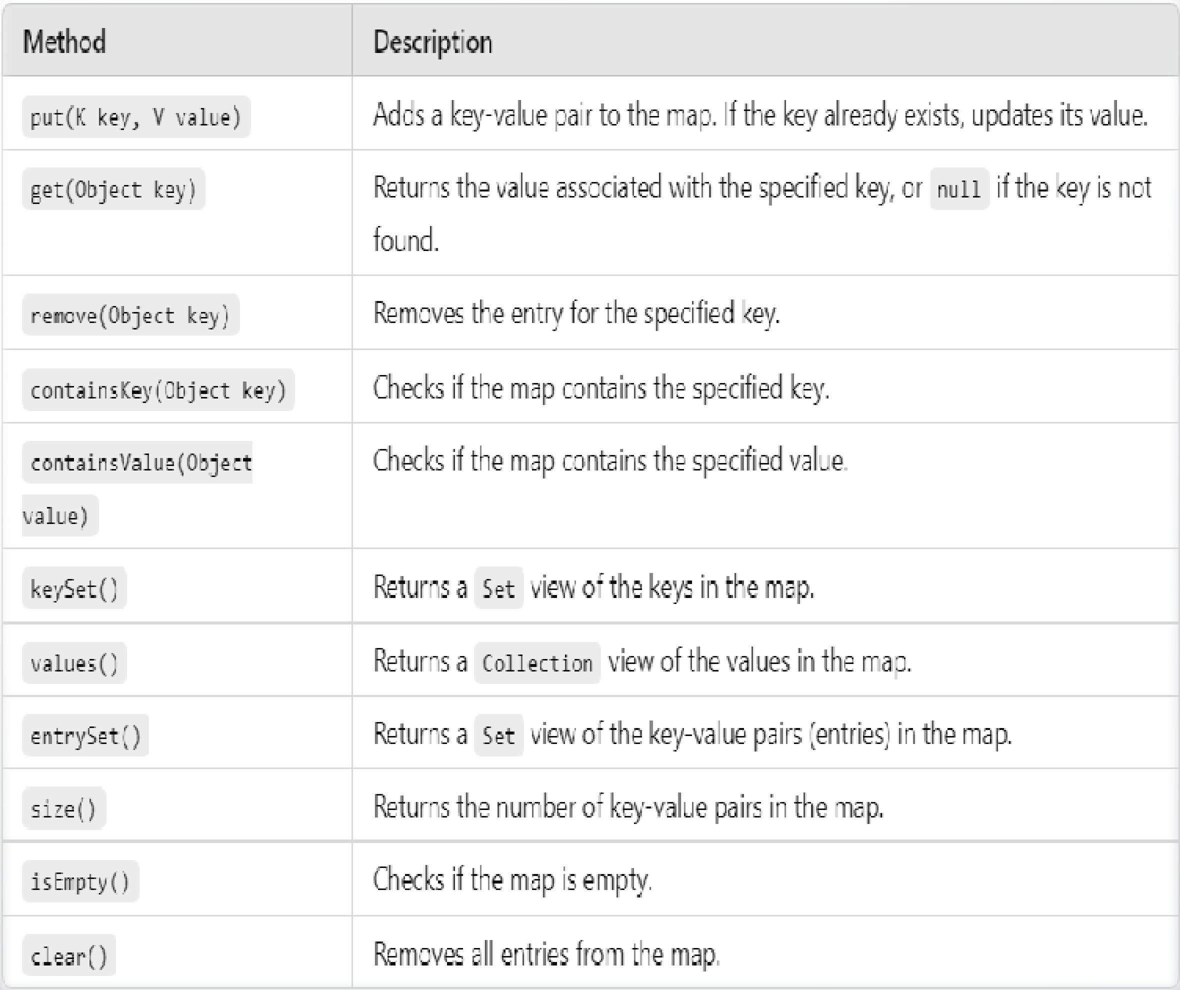
- ❑ HashMap
- ❑ LinkedHashMap
- ❑ TreeMap (implements SortedMap interface that extends Map Interface)
- ❑ WeakHashMap
- ❑ ConcurrentHashMap



Map Features & Methods

Key Features of Map Interface

- ❑ Each **key** maps to exactly one **value**. **Keys** must be unique, while **values** can be duplicated.
- ❑ The order of keys and values is not guaranteed in a general Map. However, specific implementations like **LinkedHashMap** maintain insertion order, and **TreeMap** maintains sorted order based on keys.
- ❑ Depending on the implementation, allows one (**HashMap** and **LinkedHashMap**) or no (**TreeMap**) null key and multiple null values.



Method	Description
`put(K key, V value)`	Adds a key-value pair to the map. If the key already exists, updates its value.
`get(Object key)`	Returns the value associated with the specified key, or `null` if the key is not found.
`remove(Object key)`	Removes the entry for the specified key.
`containsKey(Object key)`	Checks if the map contains the specified key.
`containsValue(Object value)`	Checks if the map contains the specified value.
`keySet()`	Returns a `Set` view of the keys in the map.
`values()`	Returns a `Collection` view of the values in the map.
`entrySet()`	Returns a `Set` view of the key-value pairs (entries) in the map.
`size()`	Returns the number of key-value pairs in the map.
`isEmpty()`	Checks if the map is empty.
`clear()`	Removes all entries from the map.

HashMap

- The HashMap class in **Java**, part of the java.util package, is a widely used implementation of the Map interface.
- It provides a way to store and retrieve key-value pairs with efficient performance (**O(1)**).
- Internally, it uses **a hash table** to store data.

```
import java.util.HashMap;
class HashMapDemo {
   public static void main(String[] args) {
      //Creating a HashMap
      HashMap<String, Integer> map = new HashMap<>();
      //Adding key-value pairs
      map.put("Apple", 50);   map.put("Banana", 20);
      map.put("Cherry", 30);  map.put("Orange", 70);
      //Retrieving a value
      System.out.println("Price of Apple: " + map.get("Apple"));
      //Checking for a key
      System.out.println("Contains 'Banana'? " + map.containsKey("Banana"));
      //Iterating through the map
      for(HashMap.Entry<String, Integer> entry : map.entrySet())
         System.out.println(entry.getKey() + ": " + entry.getValue());
   }
}
```

Output:

```
Price of Apple: 50
Contains 'Banana'? true
Apple: 50
Cherry: 30
Orange: 70
Banana: 20
```

LinkedHashMap

- The LinkedHashMap class is a part of the java.util package.
- It extends HashMap and implements the Map interface.
- Unlike HashMap, it maintains the insertion order of its elements using a **doubly-linked list** running through its entries.

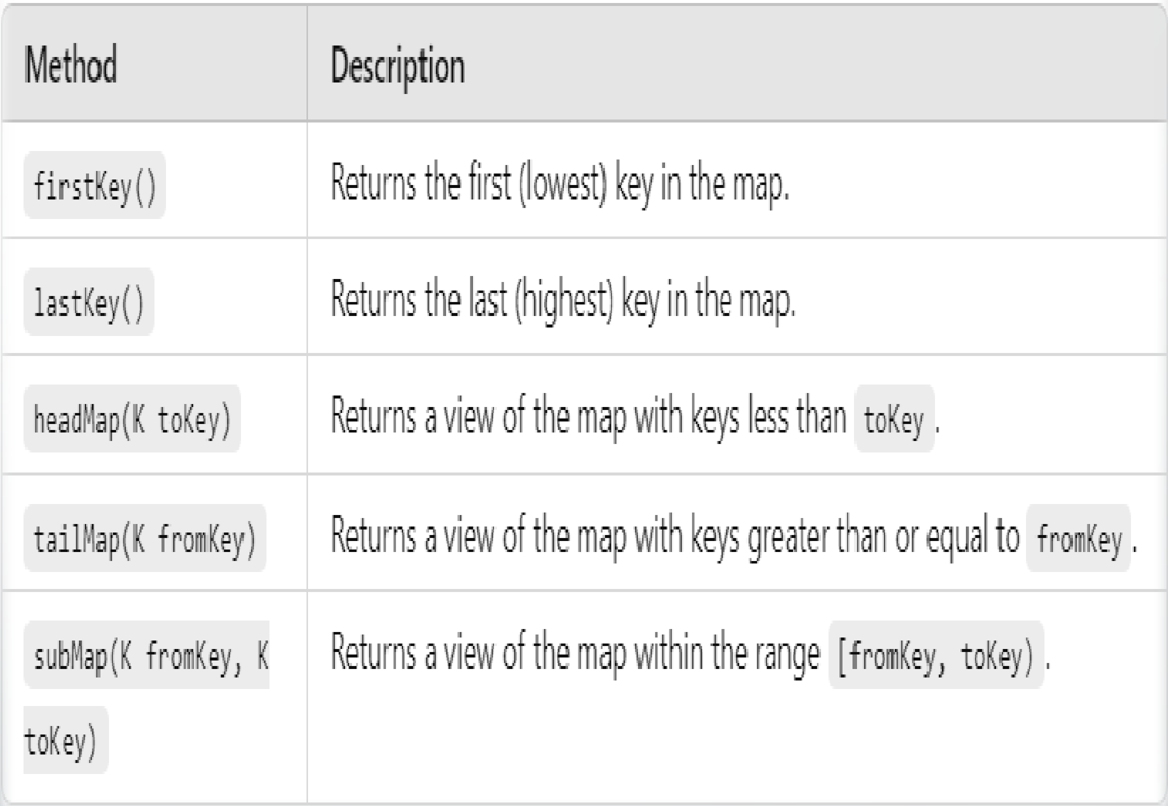
```
import java.util.LinkedHashMap;
class LinkedHashMapDemo {
   public static void main(String[] args) {
      //Creating a LinkedHashMap
      LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
      //Adding key-value pairs
      map.put("Apple", 50);   map.put("Banana", 20);
      map.put("Orange", 70);  map.put("Cherry", 30);
      //Retrieving a value
      System.out.println("Price of Apple: " + map.get("Apple"));
      //Checking for a key
      System.out.println("Contains 'Banana'? " + map.containsKey("Banana"));
      //Showing the Map
      System.out.println(map);
   }
}
```

Output:

```
Price of Apple: 50
Contains 'Banana'? true
{Apple=50,  Banana=20,
Orange=70, Cherry=30}
```

SortedMap Interface

- The SortedMap interface is a part of the java.util package.
- It is a specialized version of the Map interface that maintains its **key-value** pairs in sorted order.
- The **keys** are sorted based on their natural ordering (if the **keys** implement Comparable) or by a specified Comparator.



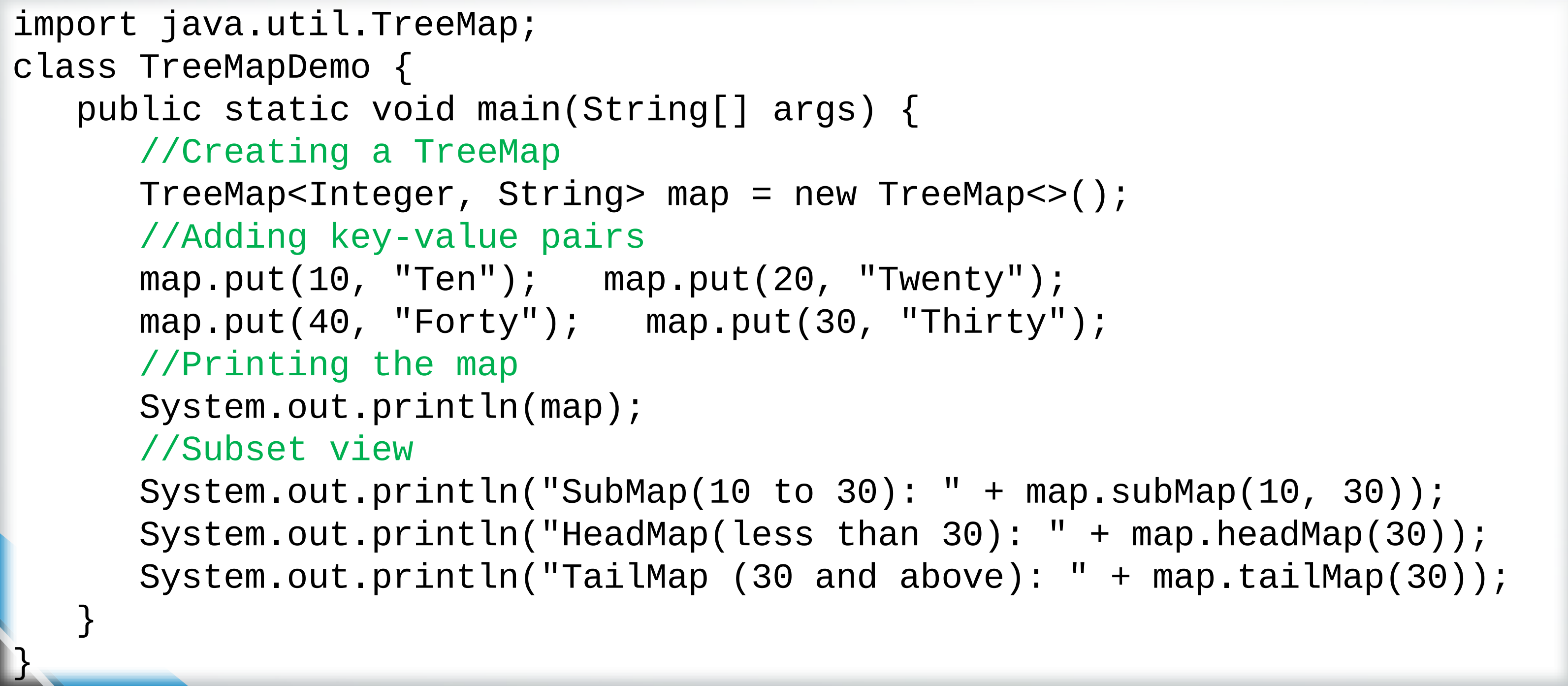
Method	Description
`firstKey()`	Returns the first (lowest) key in the map.
`lastKey()`	Returns the last (highest) key in the map.
`headMap(K toKey)`	Returns a view of the map with keys less than `toKey`.
`tailMap(K fromKey)`	Returns a view of the map with keys greater than or equal to `fromKey`.
`subMap(K fromKey, K toKey)`	Returns a view of the map within the range `[fromKey, toKey)`.

Implementation

- ❑ TreeMap
- ❑ ConcurrentSkipListMap

TreeMap

- The TreeMap class is the most commonly used implementation of the SortedMap interface.
- It is a part of the java.util package and is based on a **red-black tree** (a self-balancing **binary search tree**).
- Basic operations like **put()**, **get()**, **remove()**, and **containsKey()** have a time complexity of **O(log n)**.



```
import java.util.TreeMap;
class TreeMapDemo {
   public static void main(String[] args) {
      //Creating a TreeMap
      TreeMap<Integer, String> map = new TreeMap<>();
      //Adding key-value pairs
      map.put(10, "Ten");   map.put(20, "Twenty");
      map.put(40, "Forty");   map.put(30, "Thirty");
      //Printing the map
      System.out.println(map);
      //Subset view
      System.out.println("SubMap(10 to 30): " + map.subMap(10, 30));
      System.out.println("HeadMap(less than 30): " + map.headMap(30));
      System.out.println("TailMap (30 and above): " + map.tailMap(30));
   }
}
```

Output:

```
{10=Ten, 20=Twenty,
30=Thirty, 40=Forty}
SubMap(10 to 30):
{10=Ten, 20=Twenty}
HeadMap(less than 30):
{10=Ten, 20=Twenty}
TailMap (30 and above):
{30=Thirty, 40=Forty}
```

Thank You!