

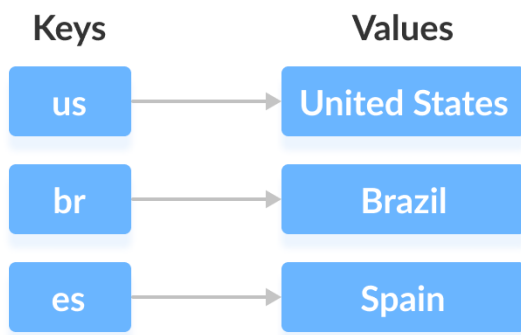
# Java Map Interface

The `Map` interface of the Java collections framework provides the functionality of the map data structure.

## Working of Map

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*.

Now, we can access those values using their corresponding keys.

**Note:** The `Map` interface maintains 3 different sets:

- the set of keys
- the set of values
- the set of key/value associations (mapping).

Hence we can access keys, values, and associations individually.

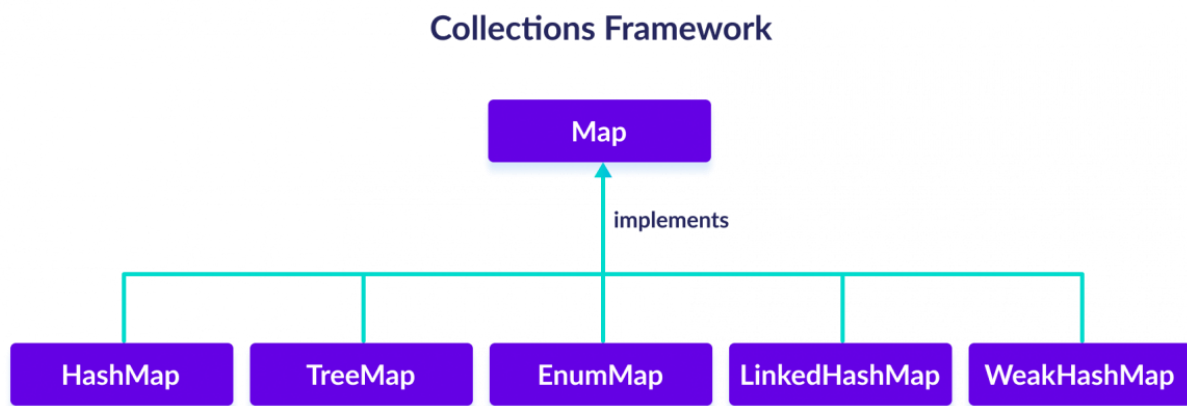
## Classes that implement Map

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

These classes are defined in the collections framework and implement the Map interface.

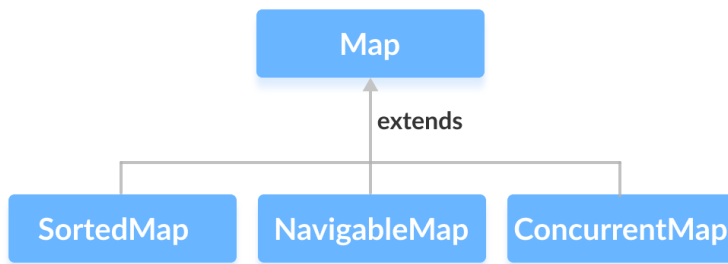


Java Map Subclasses

## Interfaces that extend Map

The Map interface is also extended by these subinterfaces:

- SortedMap
- NavigableMap
- ConcurrentMap



Java Map Subinterfaces

## How to use Map?

In Java, we must import the `java.util.Map` package in order to use `Map`. Once we import the package, here's how we can create a map.

```
// Map implementation using HashMap
Map<Key, Value> numbers = new HashMap<>();
```

In the above code, we have created a `Map` named *numbers*. We have used the `HashMap` class to implement the `Map` interface.

Here,

- **Key** - a unique identifier used to associate each element (value) in a map
- **Value** - elements associated by keys in a map

## Methods of Map

The `Map` interface includes all the methods of the `Collection` interface. It is because `Collection` is a super interface of `Map`.

Besides methods available in the `Collection` interface, the `Map` interface also includes the following methods:

- **put(K, V)** - Inserts the association of a key *K* and a value *V* into the map. If the key is already present, the new value replaces the old value.
- **putAll()** - Inserts all the entries from the specified map to this map.
- **putIfAbsent(K, V)** - Inserts the association if the key *K* is not already associated with the value *V*.
- **get(K)** - Returns the value associated with the specified key *K*. If the key is not found, it returns `null`.

- **getOrDefault(K, defaultValue)** - Returns the value associated with the specified key *K*. If the key is not found, it returns the *defaultValue*.
- **containsKey(K)** - Checks if the specified key *K* is present in the map or not.
- **containsValue(V)** - Checks if the specified value *V* is present in the map or not.
- **replace(K, V)** - Replace the value of the key *K* with the new specified value *V*.
- **replace(K, oldValue, newValue)** - Replaces the value of the key *K* with the new value *newValue* only if the key *K* is associated with the value *oldValue*.
- **remove(K)** - Removes the entry from the map represented by the key *K*.
- **remove(K, V)** - Removes the entry from the map that has key *K* associated with value *V*.
- **keySet()** - Returns a set of all the keys present in a map.
- **values()** - Returns a set of all the values present in a map.
- **entrySet()** - Returns a set of all the key/value mapping present in a map.

## Implementation of the Map Interface

### 1. Implementing HashMap Class

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

class Main {

    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
```

## 2. Implementing TreeMap Class

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

class Main {

    public static void main(String[] args) {
        // Creating Map using TreeMap
        Map<String, Integer> values = new TreeMap<>();

        // Insert elements to map
        values.put("Second", 2);
        values.put("First", 1);
        System.out.println("Map using TreeMap: " + values);

        // Replacing the values
        values.replace("First", 11);
        values.replace("Second", 22);
        System.out.println("New Map: " + values);

        // Remove elements from the map
        int removedValue = values.remove("First");
        System.out.println("Removed Value: " + removedValue);
    }
}
```

## Output

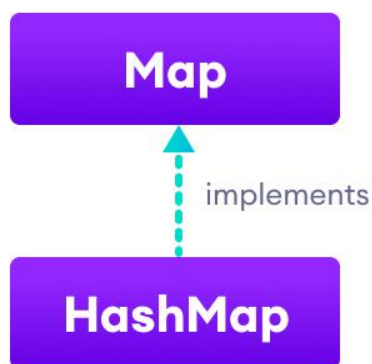
```
Map using TreeMap: {First=1, Second=2}
New Map: {First=11, Second=22}
Removed Value: 11
```

# Java HashMap

The HashMap class of the Java collections framework provides the functionality of the hash table data structure.

It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map.

The HashMap class implements the Map interface.



Java HashMap Implementation

## Create a HashMap

In order to create a hash map, we must import the `java.util.HashMap` package first. Once we import the package, here is how we can create hashmaps in Java.

```
// hashMap creation with 8 capacity and 0.6 load factor
HashMap<K, V> numbers = new HashMap<>();
```

In the above code, we have created a hashmap named *numbers*. Here, **K** represents the key type and **V** represents the type of values. For example,

```
HashMap<String, Integer> numbers = new HashMap<>();
```

Here, the type of **keys** is String and the type of **values** is Integer.

## Example 1: Create HashMap in Java

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        // create a hashmap
        HashMap<String, Integer> languages = new HashMap<>();

        // add elements to hashmap
        languages.put("Java", 8);
        languages.put("JavaScript", 1);
        languages.put("Python", 3);
        System.out.println("HashMap: " + languages);
    }
}
```

## Output

HashMap: {Java=8, JavaScript=1, Python=3}

In the above example, we have created a HashMap named *languages*.

Here, we have used the put() method to add elements to the hashmap.

## Basic Operations on Java HashMap

The HashMap class provides various methods to perform different operations on hashmaps. We will look at some commonly used arraylist operations in this tutorial:

- Add elements
- Access elements
- Change elements
- Remove elements

### 1. Add elements to a HashMap

To add a single element to the hashmap, we use the put() method of the HashMap class. For example,

```
import java.util.HashMap;

class Main {
```

```

public static void main(String[] args) {

    // create a hashmap
    HashMap<String, Integer> numbers = new HashMap<>();

    System.out.println("Initial HashMap: " + numbers);
    // put() method to add elements
    numbers.put("One", 1);
    numbers.put("Two", 2);
    numbers.put("Three", 3);
    System.out.println("HashMap after put(): " + numbers);
}
}

```

## Output

```

Initial HashMap: {}
HashMap after put(): {One=1, Two=2, Three=3}

```

In the above example, we have created a HashMap named *numbers*. Here, we have used the `put()` method to add elements to numbers.

Notice the statement,

```
numbers.put("One", 1);
```

Here, we are passing the String value *One* as the key and Integer value *1* as the value to the `put()` method.

## 2. Access HashMap Elements

We can use the `get()` method to access the value from the hashmap. For example,

```

import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        HashMap<Integer, String> languages = new HashMap<>();
        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("HashMap: " + languages);

        // get() method to get value
        String value = languages.get(1);
        System.out.println("Value at index 1: " + value);
    }
}

```



```
}
```

## Output

```
HashMap: {1=Java, 2=Python, 3=JavaScript}  
Value at index 1: Java
```

In the above example, notice the expression,

```
languages.get(1);
```

Here, the `get()` method takes the **key** as its argument and returns the corresponding **value** associated with the key.

We can also access the **keys**, **values**, and **key/value** pairs of the hashmap as set views using `keySet()`, `values()`, and `entrySet()` methods respectively. For example,

```
import java.util.HashMap;  
  
class Main {  
    public static void main(String[] args) {  
        HashMap<Integer, String> languages = new HashMap<>();  
  
        languages.put(1, "Java");  
        languages.put(2, "Python");  
        languages.put(3, "JavaScript");  
        System.out.println("HashMap: " + languages);  
  
        // return set view of keys  
        // using keySet()  
        System.out.println("Keys: " + languages.keySet());  
  
        // return set view of values  
        // using values()  
        System.out.println("Values: " + languages.values());  
  
        // return set view of key/value pairs  
        // using entrySet()  
        System.out.println("Key/Value mappings: " + languages.entrySet());  
    }  
}
```

## Output

```
HashMap: {1=Java, 2=Python, 3=JavaScript}  
Keys: [1, 2, 3]  
Values: [Java, Python, JavaScript]  
Key/Value mappings: [1=Java, 2=Python, 3=JavaScript]
```

In the above example, we have created a hashmap named *languages*. Here, we are accessing the **keys**, **values**, and **key/value** mappings from the hashmap.

### 3. Change HashMap Value

We can use the `replace()` method to change the value associated with a key in a hashmap. For example,

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        HashMap<Integer, String> languages = new HashMap<>();
        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("Original HashMap: " + languages);

        // change element with key 2
        languages.replace(2, "C++");
        System.out.println("HashMap using replace(): " + languages);
    }
}
```

### Output

```
Original HashMap: {1=Java, 2=Python, 3=JavaScript}
HashMap using replace(): {1=Java, 2=C++, 3=JavaScript}
```

In the above example, we have created a hashmap named *languages*. Notice the expression,

```
languages.replace(2, "C++");
```

Here, we are changing the value referred to by key **2** with the new value *C++*.

### 4. Remove HashMap Elements

To remove elements from a hashmap, we can use the `remove()` method. For example,

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {
```

```

HashMap<Integer, String> languages = new HashMap<>();
languages.put(1, "Java");
languages.put(2, "Python");
languages.put(3, "JavaScript");
System.out.println("HashMap: " + languages);

// remove element associated with key 2
String value = languages.remove(2);
System.out.println("Removed value: " + value);

System.out.println("Updated HashMap: " + languages);
}
}

```

## Output

```

HashMap: {1=Java, 2=Python, 3=JavaScript}
Removed value: Python
Updated HashMap: {1=Java, 3=JavaScript}

```

Here, the `remove()` method takes the **key** as its parameter. It then returns the **value** associated with the **key** and removes the **entry**.

We can also remove the entry only under certain conditions. For example,

```
remove(2, "C++");
```

Here, the `remove()` method only removes the **entry** if the **key 2** is associated with the **value C++**. Since **2** is not associated with **C++**, it doesn't remove the entry.

## Other Methods of HashMap

Method	Description
<code>clear()</code>	removes all mappings from the HashMap
<code>compute()</code>	computes a new value for the specified key
<code>computeIfAbsent()</code>	computes value if a mapping for the key is not present
<code>computeIfPresent()</code>	computes a value for mapping if the key is present
<code>merge()</code>	merges the specified mapping to the HashMap

clone()	makes the copy of the HashMap
containsKey()	checks if the specified key is present in Hashmap
containsValue()	checks if Hashmap contains the specified value
size()	returns the number of items in HashMap
isEmpty()	checks if the Hashmap is empty

## Iterate through a HashMap

To iterate through each entry of the hashmap, we can use Java for-each loop. We can iterate through **keys only**, **vales only**, and **key/value mapping**. For example,

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

class Main {
    public static void main(String[] args) {

        // create a HashMap
        HashMap<Integer, String> languages = new HashMap<>();
        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("HashMap: " + languages);

        // iterate through keys only
        System.out.print("Keys: ");
        for (Integer key : languages.keySet()) {
            System.out.print(key);
            System.out.print(", ");
        }

        // iterate through values only
        System.out.print("\nValues: ");
        for (String value : languages.values()) {
            System.out.print(value);
            System.out.print(", ");
        }

        // iterate through key/value entries
        System.out.print("\nEntries: ");
        for (Entry<Integer, String> entry : languages.entrySet()) {
            System.out.print(entry);
            System.out.print(", ");
        }
    }
}
```

```
}  
}  
}
```

## Output

```
HashMap: {1=Java, 2=Python, 3=JavaScript}  
Keys: 1, 2, 3,  
Values: Java, Python, JavaScript,  
Entries: 1=Java, 2=Python, 3=JavaScript,
```

Note that we have used the `Map.Entry` in the above example. It is the nested class of the `Map` interface that returns a view (elements) of the map.

We first need to import the `java.util.Map.Entry` package in order to use this class.

This nested class returns a view (elements) of the map.

## Creating HashMap from Other Maps

In Java, we can also create a hashmap from other maps. For example,

```
import java.util.HashMap;  
import java.util.TreeMap;  
  
class Main {  
    public static void main(String[] args) {  
  
        // create a treemap  
        TreeMap<String, Integer> evenNumbers = new TreeMap<>();  
        evenNumbers.put("Two", 2);  
        evenNumbers.put("Four", 4);  
        System.out.println("TreeMap: " + evenNumbers);  
  
        // create hashmap from the treemap  
        HashMap<String, Integer> numbers = new HashMap<>(evenNumbers);  
        numbers.put("Three", 3);  
        System.out.println("HashMap: " + numbers);  
    }  
}
```

## Output

```
TreeMap: {Four=4, Two=2}  
HashMap: {Two=2, Three=3, Four=4}
```

In the above example, we have created a `TreeMap` named `evenNumbers`. Notice the expression,

```
numbers = new HashMap<>(evenNumbers)
```

Here, we are creating a `HashMap` named `numbers` using the `TreeMap`.

**Note:** While creating a hashmap, we can include optional parameters: **capacity** and **load factor**. For example,

```
HashMap<K, V> numbers = new HashMap<>(8, 0.6f);
```

Here,

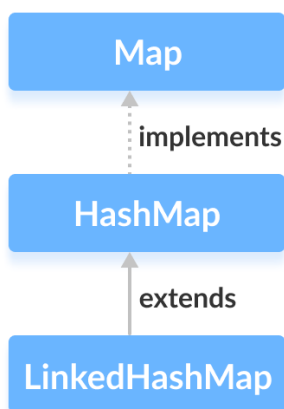
- **8** (capacity is 8) - This means it can store 8 entries.
- **0.6f** (load factor is 0.6) - This means whenever our hash table is filled by 60%, the entries are moved to a new hash table double the size of the original hash table.

If the optional parameters not used, then the default **capacity** will be **16** and the default **load factor** will be **0.75**.

## Java LinkedHashMap

The `LinkedHashMap` class of the Java collections framework provides the hash table and linked list implementation of the `Map` interface.

The `LinkedHashMap` interface extends the `HashMap` class to store its entries in a hash table. It internally maintains a doubly-linked list among all of its entries to order its entries.



## Creating a LinkedHashMap

In order to create a linked hashmap, we must import the `java.util.LinkedHashMap` package first. Once we import the package, here is how we can create linked hashmaps in Java.

```
// LinkedHashMap with initial capacity 8 and load factor 0.6
LinkedHashMap<Key, Value> numbers = new LinkedHashMap<>(8, 0.6f);
```

In the above code, we have created a linked hashmap named *numbers*.

Here,

- **Key** - a unique identifier used to associate each element (value) in a map
- **Value** - elements associated by the keys in a map

Notice the part `new LinkedHashMap<>(8, 0.6)`. Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

- **capacity** - The capacity of this linked hashmap is 8. Meaning, it can store 8 entries.
- **loadFactor** - The load factor of this linked hashmap is 0.6. This means, whenever our hash map is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

## Default capacity and load factor

It's possible to create a linked hashmap without defining its capacity and load factor. For example,

```
//LinkedHashMap with default capacity and load factor
LinkedHashMap<Key, Value> numbers1 = new LinkedHashMap<>();
```

By default,

- the capacity of the linked hashmap will be 16
- the load factor will be 0.75

**Note:** The `LinkedHashMap` class also allows us to define the order of its entries. For example

```
// LinkedHashMap with specified order
```

```
LinkedHashMap<Key, Value> numbers2 = new LinkedHashMap<>(capacity,  
loadFactor, accessOrder);
```

Here, `accessOrder` is a boolean value. Its default value is `false`. In this case entries in the linked hashmap are ordered on the basis of their insertion order.

However, if `true` is passed as `accessOrder`, entries in the linked hashmap will be ordered from least-recently accessed to most-recently accessed.

## Creating LinkedHashMap from Other Maps

Here is how we can create a linked hashmap containing all the elements of other maps.

```
import java.util.LinkedHashMap;  
  
class Main {  
    public static void main(String[] args) {  
        // Creating a LinkedHashMap of even numbers  
        LinkedHashMap<String, Integer> evenNumbers = new  
LinkedHashMap<>();  
        evenNumbers.put("Two", 2);  
        evenNumbers.put("Four", 4);  
        System.out.println("LinkedHashMap1: " + evenNumbers);  
  
        // Creating a LinkedHashMap from other LinkedHashMap  
        LinkedHashMap<String, Integer> numbers = new  
LinkedHashMap<>(evenNumbers);  
        numbers.put("Three", 3);  
        System.out.println("LinkedHashMap2: " + numbers);  
    }  
}
```

## Output

```
LinkedHashMap1: {Two=2, Four=4}  
LinkedHashMap2: {Two=2, Four=4, Three=3}
```



## Methods of LinkedHashMap

The `LinkedHashMap` class provides methods that allow us to perform various operations on the map.

### Insert Elements to LinkedHashMap

- **`put()`** - inserts the specified key/value mapping to the map
- **`putAll()`** - inserts all the entries from the specified map to this map
- **`putIfAbsent()`** - inserts the specified key/value mapping to the map if the specified key is not present in the map

For example,

```
import java.util.LinkedHashMap;

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

        // Using put()
        evenNumbers.put("Two", 2);
        evenNumbers.put("Four", 4);
        System.out.println("Original LinkedHashMap: " + evenNumbers);

        // Using putIfAbsent()
        evenNumbers.putIfAbsent("Six", 6);
        System.out.println("Updated LinkedHashMap(): " + evenNumbers);

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

        // Using putAll()
        numbers.putAll(evenNumbers);
        System.out.println("New LinkedHashMap: " + numbers);
    }
}
```

### Output

```
Original LinkedHashMap: {Two=2, Four=4}
Updated LinkedHashMap: {Two=2, Four=4, Six=6}
```

New LinkedHashMap: {One=1, Two=2, Four=4, Six=6}

## Access LinkedHashMap Elements

### 1. Using entrySet(), keySet() and values()

- **entrySet()** - returns a set of all the key/value mapping of the map
- **keySet()** - returns a set of all the keys of the map
- **values()** - returns a set of all the values of the map

For example,

```
import java.util.LinkedHashMap;

class Main {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> numbers = new
LinkedHashMap<>();

        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("LinkedHashMap: " + numbers);

        // 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());
    }
}
```

### Output

```
LinkedHashMap: {One=1, Two=2, Three=3}
Key/Value mappings: [One=1, Two=2, Three=3]
Keys: [One, Two, Three]
Values: [1, 2, 3]
```

## 2. Using `get()` and `getOrDefault()`

- **`get()`** - Returns the value associated with the specified key. If the key is not found, it returns `null`.
- **`getOrDefault()`** - Returns the value associated with the specified key. If the key is not found, it returns the specified default value.

For example,

```
import java.util.LinkedHashMap;

class Main {
    public static void main(String[] args) {

        LinkedHashMap<String, Integer> numbers = new
LinkedHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("LinkedHashMap: " + numbers);

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

        // Using getOrDefault()
        int value2 = numbers.getOrDefault("Five", 5);
        System.out.println("Returned Number: " + value2);
    }
}
```

### Output

```
LinkedHashMap: {One=1, Two=2, Three=3}
Returned Number: 3
Returned Number: 5
```

## Removed `LinkedHashMap` Elements

- **`remove(key)`** - returns and removes the entry associated with the specified *key* from the map
- **`remove(key, value)`** - removes the entry from the map only if the specified *key* mapped to be the specified *value* and return a boolean value

For example,

```
import java.util.LinkedHashMap;

class Main {
    public static void main(String[] args) {

        LinkedHashMap<String, Integer> numbers = new
LinkedHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("LinkedHashMap: " + numbers);

        // 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 removed? " + result);

        System.out.println("Updated LinkedHashMap: " + numbers);
    }
}
```

## Output

```
LinkedHashMap: {One=1, Two=2, Three=3}
Removed value: 2
Is the entry {Three=3} removed? True
Updated LinkedHashMap: {One=1}
```

## Other Methods of LinkedHashMap

Method	Description
<code>clear()</code>	removes all the entries from the map
<code>containsKey()</code>	checks if the map contains the specified key and returns a boolean value
<code>containsValue()</code>	checks if the map contains the specified value and returns a boolean value
<code>size()</code>	returns the size of the map
<code>isEmpty()</code>	checks if the map is empty and returns a boolean value

## LinkedHashMap Vs. HashMap

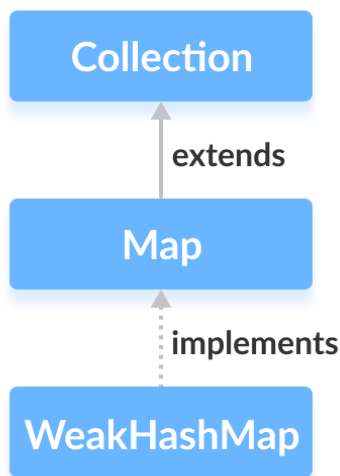
Both the LinkedHashMap and the HashMap implements the Map interface. However, there exist some differences between them.

- LinkedHashMap maintains a doubly-linked list internally. Due to this, it maintains the insertion order of its elements.
- The LinkedHashMap class requires more storage than HashMap. This is because LinkedHashMap maintains linked lists internally.
- The performance of LinkedHashMap is slower than HashMap.

## Java WeakHashMap

The WeakHashMap class of the Java collections framework provides the feature of the hash table data structure..

It implements the Map interface.



**Note:** Keys of the weak hashmap are of the **WeakReference** type.

The object of a weak reference type can be garbage collected in Java if the reference is no longer used in the program.

## Create a WeakHashMap

In order to create a weak hashmap, we must import the `java.util.WeakHashMap` package first. Once we import the package, here is how we can create weak hashmaps in Java.

```
//WeakHashMap creation with capacity 8 and load factor 0.6
WeakHashMap<Key, Value> numbers = new WeakHashMap<>(8, 0.6);
```

In the above code, we have created a weak hashmap named *numbers*.

Here,

- **Key** - a unique identifier used to associate each element (value) in a map
- **Value** - elements associated by keys in a map

Notice the part `new WeakHashMap<>(8, 0.6)`. Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

- **capacity** - The capacity of this map is 8. Meaning, it can store 8 entries.
- **loadFactor** - The load factor of this map is 0.6. This means whenever our hash table is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

## Default capacity and load factor

It is possible to create a weak hashmap without defining its capacity and load factor. For example,

```
// WeakHashMap with default capacity and load factor
WeakHashMap<Key, Value> numbers1 = new WeakHashMap<>();
```

By default,

- the capacity of the map will be 16
- the load factor will be 0.75

## Differences Between HashMap and WeakHashMap

Let us see the implementation of a weak hashmap in Java.

```
import java.util.WeakHashMap;

class Main {
    public static void main(String[] args) {
        // Creating WeakHashMap of numbers
        WeakHashMap<String, Integer> numbers = new WeakHashMap<>();

        String two = new String("Two");
        Integer twoValue = 2;
        String four = new String("Four");
        Integer fourValue = 4;

        // Inserting elements
        numbers.put(two, twoValue);
        numbers.put(four, fourValue);
        System.out.println("WeakHashMap: " + numbers);

        // Make the reference null
        two = null;

        // Perform garbage collection
        System.gc();

        System.out.println("WeakHashMap after garbage collection: " +
            numbers);
    }
}
```

### Output

```
WeakHashMap: {Four=4, Two=2}
WeakHashMap after garbage collection: {Four}
```

As we can see, when the key *two* of a weak hashmap is set to null and perform garbage collection, the key is removed.

It is because unlike hashmaps, keys of weak hashmaps are of **weak reference** type. This means the entry of a map are removed by the garbage collector if the key to that entry is no longer used. This is useful to save resources.

Now let us see the same implementation in a hashmap.

```

import java.util.HashMap;

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

        String two = new String("Two");
        Integer twoValue = 2;
        String four = new String("Four");
        Integer fourValue = 4;

        // Inserting elements
        numbers.put(two, twoValue);
        numbers.put(four, fourValue);
        System.out.println("HashMap: " + numbers);

        // Make the reference null
        two = null;

        // Perform garbage collection
        System.gc();

        System.out.println("HashMap after garbage collection: " +
numbers);
    }
}

```

## Output

```

HashMap: {Four=4, Two=2}
HashMap after garbage collection: {Four=4, Two=2}

```

Here, when the key *two* of the hashmap is set to null and perform garbage collection, the key is not removed.

This is because unlike weak hashmaps keys of hashmaps are of **strong reference** type. This means the entry of a map is not removed by the garbage collector even though the key to that entry is no longer used.

**Note:** All functionalities of hashmaps and weak hashmaps are similar except keys of a weak hashmap are of weak reference, whereas keys of a hashmap are of strong reference.



## Creating WeakHashMap from Other Maps

Here is how we can create a weak hashmap from other maps.

```
import java.util.HashMap;
import java.util.WeakHashMap;

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

        String two = new String("Two");
        Integer twoValue = 2;
        evenNumbers.put(two, twoValue);
        System.out.println("HashMap: " + evenNumbers);

        // Creating a weak hash map from other hashmap
        WeakHashMap<String, Integer> numbers = new
WeakHashMap<>(evenNumbers);

        System.out.println("WeakHashMap: " + numbers);
    }
}
```

### Output

```
HashMap: {Two=2}
WeakHashMap: {Two=2}
```

## Methods of WeakHashMap

The `WeakHashMap` class provides methods that allow us to perform various operations on the map.

### Insert Elements to WeakHashMap

- **put()** - inserts the specified key/value mapping to the map
- **putAll()** - inserts all the entries from specified map to this map
- **putIfAbsent()** - inserts the specified key/value mapping to the map if the specified key is not present in the map

For example,

```
import java.util.WeakHashMap;

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

        String two = new String("Two");
        Integer twoValue = 2;

        // Using put()
        evenNumbers.put(two, twoValue);

        String four = new String("Four");
        Integer fourValue = 4;

        // Using putIfAbsent()
        evenNumbers.putIfAbsent(four, fourValue);
        System.out.println("WeakHashMap of even numbers: " +
evenNumbers);

        //Creating WeakHashMap of numbers
        WeakHashMap<String, Integer> numbers = new WeakHashMap<>();

        String one = new String("One");
        Integer oneValue = 1;
        numbers.put(one, oneValue);

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

## Output

```
WeakHashMap of even numbers: {Four=4, Two=2}
WeakHashMap of numbers: {Two=2, Four=4, One=1}
```

## Access WeakHashMap Elements

### 1. Using entrySet(), keySet() and values()

- **entrySet()** - returns a set of all the key/value mapping of the map
- **keySet()** - returns a set of all the keys of the map
- **values()** - returns a set of all the values of the map

For example,

```
import java.util.WeakHashMap;

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

        String one = new String("One");
        Integer oneValue = 1;
        numbers.put(one, oneValue);

        String two = new String("Two");
        Integer twoValue = 2;
        numbers.put(two, twoValue);

        System.out.println("WeakHashMap: " + numbers);

        // 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());
    }
}
```

## Output

```
WeakHashMap: {Two=2, One=1}
Key/Value mappings: [Two=2, One=1]
Keys: [Two, One]
Values: [1, 2]
```

## 2. Using `get()` and `getOrDefault()`

- **`get()`** - Returns the value associated with the specified key. Returns `null` if the key is not found.
- **`getOrDefault()`** - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

For example,

```
import java.util.WeakHashMap;

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

        String one = new String("One");
        Integer oneValue = 1;
        numbers.put(one, oneValue);

        String two = new String("Two");
        Integer twoValue = 2;
        numbers.put(two, twoValue);

        System.out.println("WeakHashMap: " + numbers);

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

        // Using getOrDefault()
        int value2 = numbers.getOrDefault("Four", 4);
        System.out.println("Using getOrDefault(): " + value2);
    }
}
```

## Output

```
WeakHashMap: {Two=2, One=1}
Using get(): 2
Using getOrDefault(): 4
```

## Remove WeakHashMap Elements

- `remove(key)` - returns and removes the entry associated with the specified key from the map
- `remove(key, value)` - removes the entry from the map only if the specified key mapped to the specified value and return a boolean value

For example,

```
import java.util.WeakHashMap;

class Main {
    public static void main(String[] args) {
```

```

// Creating WeakHashMap of even numbers
WeakHashMap<String, Integer> numbers = new WeakHashMap<>();

String one = new String("One");
Integer oneValue = 1;
numbers.put(one, oneValue);

String two = new String("Two");
Integer twoValue = 2;
numbers.put(two, twoValue);

System.out.println("WeakHashMap: " + numbers);

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

// Using remove() with 2 parameters
boolean result = numbers.remove("One", 3);
System.out.println("Is the entry {One=3} removed? " + result);

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

```

## Output

```

WeakHashMap: {Two=2, One=1}
Removed value: 2
Is the entry {One=3} removed? False
Updated WeakHashMap: {One=1}

```

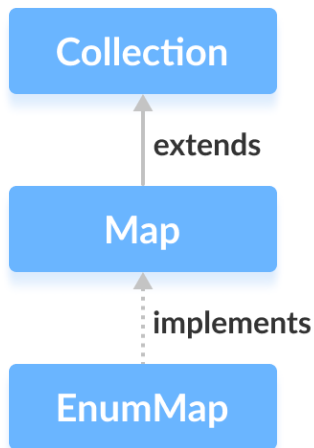
## Other Methods of WeakHashMap

Method	Description
<code>clear()</code>	Removes all the entries from the map
<code>containsKey()</code>	Checks if the map contains the specified key and returns a boolean value
<code>containsValue()</code>	Checks if the map contains the specified value and returns a boolean value
<code>size()</code>	Returns the size of the map
<code>isEmpty()</code>	Checks if the map is empty and returns a boolean value

# Java EnumMap

The EnumMap class of the Java collections framework provides a map implementation for elements of an enum.

In EnumMap, enum elements are used as **keys**. It implements the Map interface.



## Creating an EnumMap

In order to create an enum map, we must import the `java.util.EnumMap` package first. Once we import the package, here is how we can create enum maps in Java.

```
enum Size {  
    SMALL, MEDIUM, LARGE, EXTRALARGE  
}  
  
EnumMap<Size, Integer> sizes = new EnumMap<>(Size.class);
```

In the above example, we have created an enum map named *sizes*.

Here,

- *Size* - **keys** of the enum that map to values
- *Integer* - **values** of the enum map associated with the corresponding keys

## Methods of EnumMap

The EnumMap class provides methods that allow us to perform various elements on the enum maps.

### Insert Elements to EnumMap

- **put()** - inserts the specified key/value mapping (entry) to the enum map
- **putAll()** - inserts all the entries of a specified map to this map

For example,

```
import java.util.EnumMap;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumMap of the Size enum
        EnumMap<Size, Integer> sizes1 = new EnumMap<>(Size.class);

        // Using the put() Method
        sizes1.put(Size.SMALL, 28);
        sizes1.put(Size.MEDIUM, 32);
        System.out.println("EnumMap1: " + sizes1);

        EnumMap<Size, Integer> sizes2 = new EnumMap<>(Size.class);

        // Using the putAll() Method
        sizes2.putAll(sizes1);
        sizes2.put(Size.LARGE, 36);
        System.out.println("EnumMap2: " + sizes2);
    }
}
```

### Output

```
EnumMap1: {SMALL=28, MEDIUM=32}
EnumMap2: {SMALL=28, MEDIUM=32, LARGE=36}
```

In the above example, we have used the putAll() method to insert all the elements of an enum map *sizes1* to an enum map of *sizes2*.

It is also possible to insert elements from other maps such as HashMap, TreeMap, etc. to an EnumMap using putAll(). However, all maps should be of the same Enum type.

## Access EnumMap Elements

### 1. Using entrySet(), keySet() and values()

- **entrySet()** - returns a set of all the keys/values mapping (entry) of an EnumMap
- **keySet()** - returns a set of all the keys of an EnumMap
- **values()** - returns a set of all the values of an EnumMap

For example,

```
import java.util.EnumMap;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumMap of the Size enum
        EnumMap<Size, Integer> sizes = new EnumMap<>(Size.class);
        sizes.put(Size.SMALL, 28);
        sizes.put(Size.MEDIUM, 32);
        sizes.put(Size.LARGE, 36);
        sizes.put(Size.EXTRALARGE, 40);
        System.out.println("EnumMap: " + sizes);

        // Using the entrySet() Method
        System.out.println("Key/Value mappings: " + sizes.entrySet());

        // Using the keySet() Method
        System.out.println("Keys: " + sizes.keySet());

        // Using the values() Method
        System.out.println("Values: " + sizes.values());
    }
}
```

## Output

```
EnumMap: {SMALL=28, MEDIUM=32, LARGE=36, EXTRALARGE=40}
Key/Value mappings: [SMALL=28, MEDIUM=32, LARGE=36, EXTRALARGE=40]
Keys: [SMALL, MEDIUM, LARGE, EXTRALARGE]
Values: [28, 32, 36, 40]
```



## 2. Using the get() Method

The `get()` method returns the value associated with the specified key. It returns `null` if the specified key is not found.

For example,

```
import java.util.EnumMap;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumMap of the Size enum
        EnumMap<Size, Integer> sizes = new EnumMap<>(Size.class);
        sizes.put(Size.SMALL, 28);
        sizes.put(Size.MEDIUM, 32);
        sizes.put(Size.LARGE, 36);
        sizes.put(Size.EXTRALARGE, 40);
        System.out.println("EnumMap: " + sizes);

        // Using the get() Method
        int value = sizes.get(Size.MEDIUM);
        System.out.println("Value of MEDIUM: " + value);
    }
}
```

## Output

```
EnumMap: {SMALL=28, MEDIUM=32, LARGE=36, EXTRALARGE=40}
Value of MEDIUM: 32
```

## Remove EnumMap Elements

- **remove(key)** - returns and removes the entry associated with the specified key from the map
- **remove(key, value)** - removes the entry from the map only if the specified key mapped to the specified value and return a boolean value

For example,

```
import java.util.EnumMap;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumMap of the Size enum
        EnumMap<Size, Integer> sizes = new EnumMap<>(Size.class);
        sizes.put(Size.SMALL, 28);
        sizes.put(Size.MEDIUM, 32);
        sizes.put(Size.LARGE, 36);
        sizes.put(Size.EXTRALARGE, 40);
        System.out.println("EnumMap: " + sizes);

        // Using the remove() Method
        int value = sizes.remove(Size.MEDIUM);
        System.out.println("Removed Value: " + value);

        boolean result = sizes.remove(Size.SMALL, 28);
        System.out.println("Is the entry {SMALL=28} removed? " +
result);

        System.out.println("Updated EnumMap: " + sizes);
    }
}
```

## Output

```
EnumMap: {SMALL=28, MEDIUM=32, LARGE=36, EXTRALARGE=40}
Removed Value: 32
Is the entry {SMALL=28} removed? True
Updated EnumMap: {LARGE=36, EXTRALARGE=40}
```

## Replace EnumMap Elements

- **replace(key, value)** - replaces the value associated with the specified *key* by the *new value*
- **replace(key, old, new)** - replaces the *old* value with the *new* value only if the *old* value is already associated with the specified *key*
- **replaceAll(function)** - replaces each value of the map with the result of the specified *function*.

```

import java.util.EnumMap;

class Main {

    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }

    public static void main(String[] args) {

        // Creating an EnumMap of the Size enum
        EnumMap<Size, Integer> sizes = new EnumMap<>(Size.class);
        sizes.put(Size.SMALL, 28);
        sizes.put(Size.MEDIUM, 32);
        sizes.put(Size.LARGE, 36);
        sizes.put(Size.EXTRALARGE, 40);
        System.out.println("EnumMap: " + sizes);

        // Using the replace() Method
        sizes.replace(Size.MEDIUM, 30);
        sizes.replace(Size.LARGE, 36, 34);
        System.out.println("EnumMap using replace(): " + sizes);

        // Using the replaceAll() Method
        sizes.replaceAll((key, oldValue) -> oldValue + 3);
        System.out.println("EnumMap using replaceAll(): " + sizes);
    }
}

```

## Output

```

EnumMap: {SMALL=28, MEDIUM=32, LARGE=36, EXTRALARGE=40}
EnumMap using replace(): {SMALL=28, MEDIUM=30, LARGE=34,
EXTRALARGE=40}
EnumMap using replaceAll(): {SMALL=31, MEDIUM=33, LARGE=37,
EXTRALARGE=43}

```

In the above program, notice the statement

```
sizes.replaceAll((key, oldValue) -> oldValue + 3);
```

Here, the method accesses all the entries of the map. It then replaces all the values with the new values provided by the lambda expressions.

## Other Methods

Method	Description
<code>clone()</code>	Creates a copy of the <code>EnumMap</code>
<code>containsKey()</code>	Searches the <code>EnumMap</code> for the specified key and returns a boolean result
<code>containsValue()</code>	Searches the <code>EnumMap</code> for the specified value and returns a boolean result
<code>size()</code>	Returns the size of the <code>EnumMap</code>
<code>clear()</code>	Removes all the entries from the <code>EnumMap</code>

## EnumSet Vs. EnumMap

Both the `EnumSet` and `EnumMap` class provides data structures to store enum values. However, there exist some major differences between them.

- Enum set is represented internally as a sequence of bits, whereas the enum map is represented internally as arrays.
- Enum set is created using its predefined methods like `allOf()`, `noneOf()`, `of()`, etc. However, an enum map is created using its constructor.

## Cloneable and Serializable Interfaces

The `EnumMap` class also implements `Cloneable` and `Serializable` interfaces.

### Cloneable Interface

It allows the `EnumMap` class to make a copy of instances of the class.

### Serializable Interface

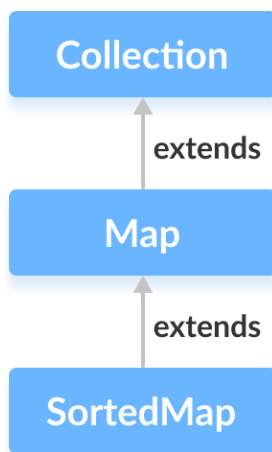
Whenever Java objects need to be transmitted over a network, objects need to be converted into bits or bytes. This is because Java objects cannot be transmitted over the network.

The `Serializable` interface allows classes to be serialized. This means objects of the classes implementing `Serializable` can be converted into bits or bytes.

## Java SortedMap Interface

The `SortedMap` interface of the Java collections framework provides sorting of keys stored in a map.

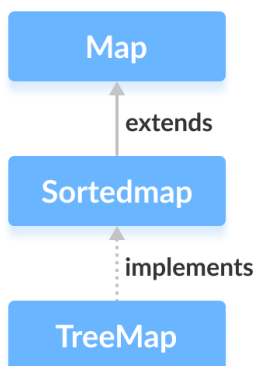
It extends the `Map` interface.



## Class that implements SortedMap

Since `SortedMap` is an interface, we cannot create objects from it.

In order to use the functionalities of the `SortedMap` interface, we need to use the class `TreeMap` that implements it.



## How to use SortedMap?

To use the `SortedMap`, we must import the `java.util.SortedMap` package first. Once we import the package, here's how we can create a sorted map.

```
// SortedMap implementation by TreeMap class
SortedMap<Key, Value> numbers = new TreeMap<>();
```

We have created a sorted map called *numbers* using the `TreeMap` class.

Here,

- **Key** - a unique identifier used to associate each element (value) in a map
- **Value** - elements associated by keys in a map

Here, we have used no arguments to create a sorted map. Hence the map will be sorted naturally (ascending order).

## Methods of SortedMap

The `SortedMap` interface includes all the methods of the `Map` interface. It is because `Map` is a super interface of `SortedMap`.

Besides all those methods, here are the methods specific to the `SortedMap` interface.

- **comparator()** - returns a comparator that can be used to order keys in a map
- **firstKey()** - returns the first key of the sorted map
- **lastKey()** - returns the last key of the sorted map
- **headMap(key)** - returns all the entries of a map whose keys are less than the specified *key*
- **tailMap(key)** - returns all the entries of a map whose keys are greater than or equal to the specified *key*
- **subMap(key1, key2)** - returns all the entries of a map whose keys lie in between *key1* and *key2* including *key1*

## Implementation of SortedMap in TreeMap Class

```
import java.util.SortedMap;
import java.util.TreeMap;

class Main {

    public static void main(String[] args) {
        // Creating SortedMap using TreeMap
        SortedMap<String, Integer> numbers = new TreeMap<>();

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

        // Access the first key of the map
        System.out.println("First Key: " + numbers.firstKey());

        // Access the last key of the map
        System.out.println("Last Key: " + numbers.lastKey());

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

## Output

```
SortedMap: {One=1, Two=2}
First Key: One
Last Key: Two
Removed Value: 1
```

Here, we show how the SortedMap interface works.

# Java NavigableMap Interface

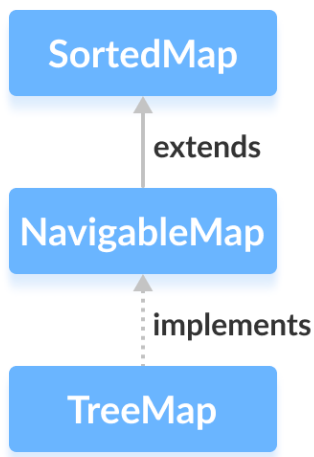
The NavigableMap interface of the Java collections framework provides the features to navigate among the map entries.

It is considered as a type of SortedMap.

## Class that implements NavigableMap

Since NavigableMap is an interface, we cannot create objects from it.

In order to use the functionalities of the NavigableMap interface, we need to use the TreeMap class that implements NavigableMap.



## How to use NavigableMap?

In Java, we must import the `java.util.NavigableMap` package to use NavigableMap. Once we import the package, here's how we can create a navigable map.

```
// NavigableMap implementation by TreeMap class
NavigableMap<Key, Value> numbers = new TreeMap<>();
```

In the above code, we have created a navigable map named *numbers* of the TreeMap class.

Here,



- **Key** - a unique identifier used to associate each element (value) in a map
- **Value** - elements associated by keys in a map

## Methods of NavigableMap

The NavigableMap is considered as a type of SortedMap. It is because NavigableMap extends the SortedMap interface.

Hence, all SortedMap methods are also available in NavigableMap.

However, some of the methods of SortedMap (headMap(), tailMap(), and subMap() ) are defined differently in NavigableMap.

Let's see how these methods are defined in NavigableMap.

**headMap(key, booleanValue)**

The headMap() method returns all the entries of a navigable map associated with all those keys before the specified *key* (which is passed as an argument).

The *booleanValue* is an optional parameter. Its default value is false.

If true is passed as a *booleanValue*, the method returns all the entries associated with all those keys before the specified *key*, including the entry associated with the specified *key*.

**tailMap(key, booleanValue)**

The tailMap() method returns all the entries of a navigable map associated with all those keys after the specified *key* (which is passed as an argument) including the entry associated with the specified *key*.

The *booleanValue* is an optional parameter. Its default value is true.

If false is passed as a *booleanValue*, the method returns all the entries associated with those keys after the specified *key*, without including the entry associated with the specified *key*.

subMap(k1, bv1, k2, bv2)

The subMap() method returns all the entries associated with keys between k1 and k2 including the entry associated with k1.

The *bv1* and *bv2* are optional parameters. The default value of *bv1* is true and the default value of *bv2* is false.

If false is passed as *bv1*, the method returns all the entries associated with keys between *k1* and *k2*, without including the entry associated with *k1*.

If true is passed as *bv2*, the method returns all the entries associated with keys between *k1* and *k2*, including the entry associated with *k1*.

## Other Methods

The NavigableMap provides various methods that can be used to locate the entries of maps.

- **descendingMap()** - reverse the order of entries in a map
- **descendingKeyMap()** - reverses the order of keys in a map
- **ceilingEntry()** - returns an entry with the lowest key among all those entries whose keys are greater than or equal to the specified key
- **ceilingKey()** - returns the lowest key among those keys that are greater than or equal to the specified key
- **floorEntry()** - returns an entry with the highest key among all those entries whose keys are less than or equal to the specified key
- **floorKey()** - returns the highest key among those keys that are less than or equal to the specified key
- **higherEntry()** - returns an entry with the lowest key among all those entries whose keys are greater than the specified key
- **higherKey()** - returns the lowest key among those keys that are greater than the specified key
- **lowerEntry()** - returns an entry with the highest key among all those entries whose keys are less than the specified key
- **lowerKey()** - returns the highest key among those keys that are less than the specified key
- **firstEntry()** - returns the first entry (the entry with the lowest key) of the map
- **lastEntry()** - returns the last entry (the entry with the highest key) of the map
- **pollFirstEntry()** - returns and removes the first entry of the map
- **pollLastEntry()** - returns and removes the last entry of the map

## Implementation of NavigableMap in TreeMap Class

```
import java.util.NavigableMap;
import java.util.TreeMap;

class Main {

    public static void main(String[] args) {
        // Creating NavigableMap using TreeMap
        NavigableMap<String, Integer> numbers = new TreeMap<>();

        // Insert elements to map
        numbers.put("Two", 2);
        numbers.put("One", 1);
        numbers.put("Three", 3);
        System.out.println("NavigableMap: " + numbers);

        // Access the first entry of the map
        System.out.println("First Entry: " + numbers.firstEntry());

        // Access the last entry of the map
        System.out.println("Last Entry: " + numbers.lastEntry());

        // Remove the first entry from the map
        System.out.println("Removed First Entry: " +
numbers.pollFirstEntry());

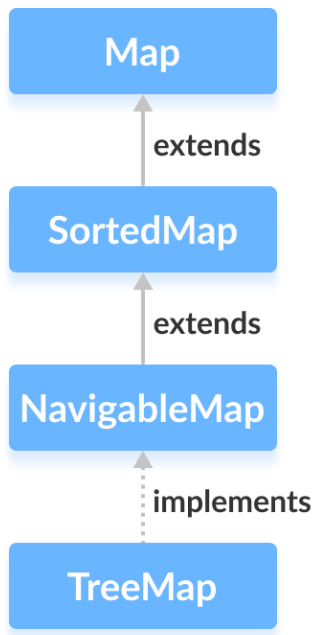
        // Remove the last entry from the map
        System.out.println("Removed Last Entry: " +
numbers.pollLastEntry());
    }
}
```

## Output

```
NavigableMap: {One=1, Three=3, Two=2}
First Entry: One=1
Last Entry: Two=2
Removed First Entry: One=1
Removed Last Entry: Two=2
```

# Java TreeMap

The TreeMap class of the Java collections framework provides the tree data structure implementation. It implements the NavigableMap interface.



## Creating a TreeMap

In order to create a TreeMap, we must import the `java.util.TreeMap` package first. Once we import the package, here is how we can create a TreeMap in Java.

```
TreeMap<Key, Value> numbers = new TreeMap<>();
```

In the above code, we have created a TreeMap named *numbers* without any arguments. In this case, the elements in TreeMap are sorted naturally (ascending order).

However, we can customize the sorting of elements by using the Comparator interface. We will learn about it later in this tutorial.

Here,

- **Key** - a unique identifier used to associate each element (value) in a map

- **Value** - elements associated by keys in a map

## Methods of TreeMap

The `TreeMap` class provides various methods that allow us to perform operations on the map.

## Insert Elements to TreeMap

- **put()** - inserts the specified key/value mapping (entry) to the map
- **putAll()** - inserts all the entries from specified map to this map
- **putIfAbsent()** - inserts the specified key/value mapping to the map if the specified key is not present in the map

For example,

```
import java.util.TreeMap;

class Main {
    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);
    }
}
```

## Output

```
TreeMap of even numbers: {Four=4, Six=6, Two=2}
TreeMap of numbers: {Four=4, One=1, Six=6, Two=2}
```

# Access TreeMap Elements

## 1. Using `entrySet()`, `keySet()` and `values()`

- **`entrySet()`** - returns a set of all the key/values mapping (entry) of a treemap
- **`keySet()`** - returns a set of all the keys of a tree map
- **`values()`** - returns a set of all the maps of a tree map

For example,

```
import java.util.TreeMap;

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

        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("TreeMap: " + numbers);

        // 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());
    }
}
```

## Output

```
TreeMap: {One=1, Three=3, Two=2}
Key/Value mappings: [One=1, Three=3, Two=2]
Keys: [One, Three, Two]
Values: [1, 3, 2]
```

## 2. Using `get()` and `getOrDefault()`

- **`get()`** - Returns the value associated with the specified key. Returns null if the key is not found.
- **`getOrDefault()`** - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

For example,

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("TreeMap: " + numbers);

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

        // Using getOrDefault()
        int value2 = numbers.getOrDefault("Five", 5);
        System.out.println("Using getOrDefault(): " + value2);
    }
}
```

## Output

```
TreeMap: {One=1, Three=3, Two=2}
Using get(): 3
Using getOrDefault(): 5
```

Here, the `getOrDefault()` method does not find the key Five. Hence it returns the specified default value 5.

## Remove TeeMap Elements

- **remove(key)** - returns and removes the entry associated with the specified key from a TreeMap
- **remove(key, value)** - removes the entry from the map only if the specified key is associated with the specified value and returns a boolean value

For example,

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

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

```

        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("TreeMap: " + numbers);

        // 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);
    }
}

```

## Output

```

TreeMap: {One=1, Three=3, Two=2}
Removed value = 2
Is the entry {Three=3} removed? True
Updated TreeMap: {One=1}

```

## Replace TreeMap Elements

- **replace(key, value)** - replaces the value mapped by the specified *key* with the new *value*
- **replace(key, old, new)** - replaces the old value with the new value only if the old value is already associated with the specified key
- **replaceAll(function)** - replaces each value of the map with the result of the specified *function*

For example,

```

import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 2);
        numbers.put("Third", 3);
        System.out.println("Original TreeMap: " + numbers);

        // Using replace()
    }
}

```



```

        numbers.replace("Second", 22);
        numbers.replace("Third", 3, 33);
        System.out.println("TreeMap using replace: " + numbers);

        // Using replaceAll()
        numbers.replaceAll((key, oldValue) -> oldValue + 2);
        System.out.println("TreeMap using replaceAll: " + numbers);
    }
}

```

## Output

```

Original TreeMap: {First=1, Second=2, Third=3}
TreeMap using replace(): {First=1, Second=22, Third=33}
TreeMap using replaceAll(): {First=3, Second=24, Third=35}

```

In the above program notice the statement

```
numbers.replaceAll((key, oldValue) -> oldValue + 2);
```

Here, we have passed a lambda expression as an argument.

The `replaceAll()` method accesses all the entries of the map. It then replaces all the elements with the new values (returned from the lambda expression).

## Methods for Navigation

Since the `TreeMap` class implements `NavigableMap`, it provides various methods to navigate over the elements of the treemap.

### 1. First and Last Methods

- **firstKey()** - returns the first key of the map
- **firstEntry()** - returns the key/value mapping of the first key of the map
- **lastKey()** - returns the last key of the map
- **lastEntry()** - returns the key/value mapping of the last key of the map

For example,

```

import java.util.TreeMap;

class Main {
    public static void main(String[] args) {
        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 2);
    }
}

```

```

        numbers.put("Third", 3);
        System.out.println("TreeMap: " + numbers);

        // 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: {First=1, Second=2, Third=3}
First Key: First
Last Key: Third
First Entry: First=1
Last Entry: Third=3

```

## 2. Ceiling, Floor, Higher and Lower Methods

- **higherKey()** - Returns the lowest key among those keys that are greater than the specified key.
- **higherEntry()** - Returns an entry associated with a key that is lowest among all those keys greater than the specified key.
- **lowerKey()** - Returns the greatest key among all those keys that are less than the specified key.
- **lowerEntry()** - Returns an entry associated with a key that is greatest among all those keys that are less than the specified key.
- **ceilingKey()** - Returns the lowest key among those keys that are greater than the specified key. If the key passed as an argument is present in the map, it returns that key.
- **ceilingEntry()** - Returns an entry associated with a key that is lowest among those keys that are greater than the specified key. If an entry associated with the key passed an argument is present in the map, it returns the entry associated with that key.
- **floorKey()** - Returns the greatest key among those keys that are less than the specified key. If the key passed as an argument is present, it returns that key.

- **floorEntry()** - Returns an entry associated with a key that is greatest among those keys that are less than the specified key. If the key passed as argument is present, it returns that key.

For example,

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 5);
        numbers.put("Third", 4);
        numbers.put("Fourth", 6);
        System.out.println("TreeMap: " + numbers);

        // Using higher()
        System.out.println("Using higherKey(): " +
numbers.higherKey("Fourth"));
        System.out.println("Using higherEntry(): " +
numbers.higherEntry("Fourth"));

        // Using lower()
        System.out.println("\nUsing lowerKey(): " +
numbers.lowerKey("Fourth"));
        System.out.println("Using lowerEntry(): " +
numbers.lowerEntry("Fourth"));

        // Using ceiling()
        System.out.println("\nUsing ceilingKey(): " +
numbers.ceilingKey("Fourth"));
        System.out.println("Using ceilingEntry(): " +
numbers.ceilingEntry("Fourth"));

        // Using floor()
        System.out.println("\nUsing floorKey(): " +
numbers.floorKey("Fourth"));
        System.out.println("Using floorEntry(): " +
numbers.floorEntry("Fourth"));

    }
}
```

## Output

```
TreeMap: {First=1, Fourth=6, Second=5, Third=4}
Using higherKey(): Second
```

Using higherEntry(): Second=5

Using lowerKey(): First

Using lowerEntry(): First=1

Using ceilingKey(): Fourth

Using ceilingEntry(): Fourth=6

Using floorkey(): Fourth

Using floorEntry(): Fourth=6

### 3. pollFirstEntry() and pollLastEntry() Methods

- **pollFirstEntry()** - returns and removes the entry associated with the first key of the map
- **pollLastEntry()** - returns and removes the entry associated with the last key of the map

For example,

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 2);
        numbers.put("Third", 3);
        System.out.println("TreeMap: " + numbers);

        //Using the pollFirstEntry() method
        System.out.println("Using pollFirstEntry(): " +
numbers.pollFirstEntry());

        // Using the pollLastEntry() method
        System.out.println("Using pollLastEntry(): " +
numbers.pollLastEntry());

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

    }
}
```

### Output

```
TreeMap: {First=1, Second=2, Third=3}
Using pollFirstEntry(): First=1
Using pollLastEntry(): Third=3
```

Updated TreeMap: {Second=2}

## 4. headMap(), tailMap() and subMap() Methods

### headMap(key, booleanValue)

The `headMap()` method returns all the key/value pairs of a treemap before the specified *key* (which is passed as an argument).

The *booleanValue* parameter is optional. Its default value is `false`.

If `true` is passed as a *booleanValue*, the method also includes the key/value pair of the *key* which is passed as an argument.

For example,

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 2);
        numbers.put("Third", 3);
        numbers.put("Fourth", 4);
        System.out.println("TreeMap: " + numbers);

        System.out.println("\nUsing headMap() Method:");
        // Using headMap() with default booleanValue
        System.out.println("Without boolean value: " +
            numbers.headMap("Fourth"));

        // Using headMap() with specified booleanValue
        System.out.println("With boolean value: " +
            numbers.headMap("Fourth", true));

    }
}
```

## Output

TreeMap: {First=1, Fourth=4, Second=2, Third=3}

Using headMap() Method:

Without boolean value: {First=1}

With boolean value: {First=1, Fourth=4}

## **tailMap(key, booleanValue)**

The `tailMap()` method returns all the key/value pairs of a treemap starting from the specified *key* (which is passed as an argument).

The *booleanValue* is an optional parameter. Its default value is `true`.

If `false` is passed as a *booleanValue*, the method doesn't include the key/value pair of the specified *key*.

For example,

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 2);
        numbers.put("Third", 3);
        numbers.put("Fourth", 4);
        System.out.println("TreeMap: " + numbers);

        System.out.println("\nUsing tailMap() Method:");
        // Using tailMap() with default booleanValue
        System.out.println("Without boolean value: " +
            numbers.tailMap("Second"));

        // Using tailMap() with specified booleanValue
        System.out.println("With boolean value: " +
            numbers.tailMap("Second", false));

    }
}
```

## **Output**

```
TreeMap: {First=1, Fourth=4, Second=2, Third=3}
```

```
Using tailMap() Method:
```

```
Without boolean value: {Second=2, Third=3}
```

```
With boolean value: {Third=3}
```

## **subMap(k1, bV1, k2, bV2)**

The subMap() method returns all the entries associated with keys between *k1* and *k2* including the entry of *k1*.

The *bV1* and *bV2* are optional boolean parameters. The default value of *bV1* is true and the default value of *bV2* is false.

If false is passed as *bV1*, the method returns all the entries associated with keys between *k1* and *k2* without including the entry of *k1*.

If true is passed as *bV2*, the method returns all the entries associated with keys between *k1* and *k2* including the entry of *k2*.

For example,

```
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 2);
        numbers.put("Third", 3);
        numbers.put("Fourth", 4);
        System.out.println("TreeMap: " + numbers);

        System.out.println("\nUsing subMap() Method:");
        // Using subMap() with default booleanValue
        System.out.println("Without boolean value: " +
            numbers.subMap("Fourth", "Third"));

        // Using subMap() with specified booleanValue
        System.out.println("With boolean value: " +
            numbers.subMap("Fourth", false, "Third", true));

    }
}
```

## **Output**

```
TreeMap: {First=1, Fourth=2, Second=2, Third=3}
```

```
Using subMap() Method:
```

```
Without boolean value: {Fourth=4, Second=2}
```

```
With boolean value: {Second=2, Third=3}
```

## Other Methods of TreeMap

Method	Description
<code>clone()</code>	Creates a copy of the <code>TreeMap</code>
<code>containsKey()</code>	Searches the <code>TreeMap</code> for the specified key and returns a boolean result
<code>containsValue()</code>	Searches the <code>TreeMap</code> for the specified value and returns a boolean result
<code>size()</code>	Returns the size of the <code>TreeMap</code>
<code>clear()</code>	Removes all the entries from the <code>TreeMap</code>

## TreeMap Comparator

In all the examples above, treemap elements are sorted naturally (in ascending order). However, we can also customize the ordering of keys.

For this, we need to create our own comparator class based on which keys in a treemap are sorted. For example,

```
        numbers.put("First", 1);
        numbers.put("Second", 2);
        numbers.put("Third", 3);
        numbers.put("Fourth", 4);
        System.out.println("TreeMap: " + numbers);
    }

    // Creating a comparator class
    public static class CustomComparator implements Comparator<String>
    {

        @Override
        public int compare(String number1, String number2) {
            int value = number1.compareTo(number2);

            // elements are sorted in reverse order
            if (value > 0) {
                return -1;
            }
            else if (value < 0) {
                return 1;
            }
            else {
```



```
        return 0;
    }
}
}
```

## Output

TreeMap: {Third=3, Second=2, Fourth=4, First=1}

In the above example, we have created a treemap passing *CustomComparator* class as an argument.

The *CustomComparator* class implements the Comparator interface.

We then override the compare() method to sort elements in reverse order.

## Java ConcurrentMap Interface

The ConcurrentMap interface of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map.

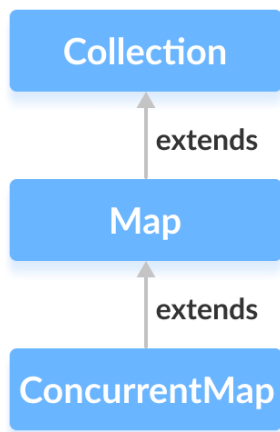
ConcurrentMap is known as a synchronized map.

It extends the Map interface.

### Class that implements ConcurrentMap

Since ConcurrentMap is an interface, we cannot create objects from it.

In order to use the functionalities of the ConcurrentMap interface, we need to use the class ConcurrentHashMap that implements it.



## How to use ConcurrentMap?

To use the ConcurrentMap, we must import the `java.util.concurrent.ConcurrentMap` package first. Once we import the package, here's how we can create a concurrent map.

```
// ConcurrentMap implementation by ConcurrentHashMap  
ConcurrentMap<Key, Value> numbers = new ConcurrentHashMap<>();
```

In the above code, we have created a concurrent map named *numbers*.

Here,

- **Key** - a unique identifier used to associate each element (value) in a map
- **Value** - elements associated by keys in a map

## Methods of ConcurrentMap

The `ConcurrentMap` interface includes all the methods of the `Map` interface. It is because `Map` is the super interface of the `ConcurrentMap` interface.

Besides all those methods, here are the methods specific to the `ConcurrentMap` interface.

- **putIfAbsent()** - Inserts the specified key/value to the map if the specified key is not already associated with any value.

- **compute()** - Computes an entry (key/value mapping) for the specified key and its previously mapped value.
- **computeIfAbsent()** - Computes a value using the specified function for the specified key if the key is not already mapped with any value.
- **computeIfPresent()** - Computes a new entry (key/value mapping) for the specified key if the key is already mapped with the specified value.
- **forEach()** - Access all entries of a map and perform the specified actions.
- **merge()** - Merges the new specified value with the old value of the specified key if the key is already mapped to a certain value. If the key is not already mapped, the method simply associates the specified value to our key.

## Implementation of ConcurrentMap in ConcurrentHashMap

```
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ConcurrentHashMap;

class Main {

    public static void main(String[] args) {
        // Creating ConcurrentMap using ConcurrentHashMap
        ConcurrentMap<String, Integer> numbers = new
        ConcurrentHashMap<>();

        // Insert elements to map
        numbers.put("Two", 2);
        numbers.put("One", 1);
        numbers.put("Three", 3);
        System.out.println("ConcurrentMap: " + numbers);

        // Access the value of specified key
        int value = numbers.get("One");
        System.out.println("Accessed Value: " + value);

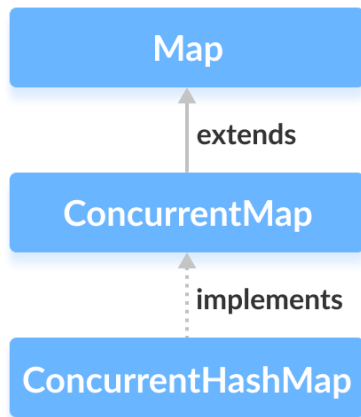
        // Remove the value of specified key
        int removedValue = numbers.remove("Two");
        System.out.println("Removed Value: " + removedValue);
    }
}
```

## Output

```
ConcurrentMap: {One=1, Two=2, Three=3}
Accessed Value: 1
Removed Value: 2
```

# Java ConcurrentHashMap

The ConcurrentHashMap class of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map. It implements the ConcurrentMap interface.



## Create a ConcurrentHashMap

In order to create a concurrent hashmap, we must import the `java.util.concurrent.ConcurrentHashMap` package first. Once we import the package, here is how we can create concurrent hashmaps in Java.

```
// ConcurrentHashMap with capacity 8 and load factor 0.6
ConcurrentHashMap<Key, Value> numbers = new ConcurrentHashMap<>(8,
0.6f);
```

In the above code, we have created a concurrent hashmap named *numbers*.

Here,

- **Key** - a unique identifier used to associate each element (value) in a map
- **Value** - elements associated by keys in a map

Notice the part `new ConcurrentHashMap<>(8, 0.6)`. Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

- **capacity** - The capacity of this map is 8. Meaning, it can store 8 entries.

- **loadFactor** - The load factor of this map is 0.6. This means, whenever our hash table is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

## Default capacity and load factor

It's possible to create a concurrent hashmap without defining its capacity and load factor. For example,

```
// ConcurrentHashMap with default capacity and load factor
ConcurrentHashMap<Key, Value> numbers1 = new ConcurrentHashMap<>();
```

By default,

- the capacity of the map will be 16
- the load factor will be 0.75

## Creating ConcurrentHashMap from Other Maps

Here is how we can create a concurrent hashmap containing all the elements of other maps.

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        // Creating a hashmap of even numbers
        HashMap<String, Integer> evenNumbers = new HashMap<>();
        evenNumbers.put("Two", 2);
        evenNumbers.put("Four", 4);
        System.out.println("HashMap: " + evenNumbers);

        // Creating a concurrent hashmap from other map
        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>(evenNumbers);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);
    }
}
```

## Output

```
HashMap: {Four=4, Two=2}
ConcurrentHashMap: {Four=4, Two=2, Three=3}
```

## Methods of ConcurrentHashMap

The `ConcurrentHashMap` class provides methods that allow us to perform various operations on the map.

### Insert Elements to ConcurrentHashMap

- **put()** - inserts the specified key/value mapping to the map
- **putAll()** - inserts all the entries from specified map to this map
- **putIfAbsent()** - inserts the specified key/value mapping to the map if the specified key is not present in the map

For example,

```
import java.util.concurrent.ConcurrentHashMap;

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

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

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

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

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

### Output

```
ConcurrentHashMap of even numbers: {Six=6, Four=4, Two=2}
ConcurrentHashMap of numbers: {Six=6, One=1, Four=4, Two=2}
```

## Access ConcurrentHashMap Elements

### 1. Using `entrySet()`, `keySet()` and `values()`

- `entrySet()` - returns a set of all the key/value mapping of the map
- `keySet()` - returns a set of all the keys of the map
- `values()` - returns a set of all the values of the map

For example,

```
import java.util.concurrent.ConcurrentHashMap;

class Main {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>();

        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);

        // 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());
    }
}
```

### Output

```
ConcurrentHashMap: {One=1, Two=2, Three=3}
Key/Value mappings: [One=1, Two=2, Three=3]
Keys: [One, Two, Three]
Values: [1, 2, 3]
```

### 2. Using `get()` and `getOrDefault()`

- **`get()`** - Returns the value associated with the specified key. Returns `null` if the key is not found.
- **`getOrDefault()`** - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

For example,

```
import java.util.concurrent.ConcurrentHashMap;
```

```

class Main {
    public static void main(String[] args) {

        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);

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

        // Using getOrDefault()
        int value2 = numbers.getOrDefault("Five", 5);
        System.out.println("Using getOrDefault(): " + value2);
    }
}

```

## Output

```

ConcurrentHashMap: {One=1, Two=2, Three=3}
Using get(): 3
Using getOrDefault(): 5

```

## Remove ConcurrentHashMap Elements

- **remove(key)** - returns and removes the entry associated with the specified key from the map
- **remove(key, value)** - removes the entry from the map only if the specified key mapped to the specified value and return a boolean value

For example,

```

import java.util.concurrent.ConcurrentHashMap;

class Main {
    public static void main(String[] args) {

        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);

        // 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 ConcurrentHashMap: " + numbers);
    }
}

```

## Output

```

ConcurrentHashMap: {One=1, Two=2, Three=3}
Removed value: 2
Is the entry {Three=3} removed? True
Updated ConcurrentHashMap: {One=1}

```

## Bulk ConcurrentHashMap Operations

The `ConcurrentHashMap` class provides different bulk operations that can be applied safely to concurrent maps.

### 1. `forEach()` Method

The `forEach()` method iterates over our entries and executes the specified function.

It includes two parameters.

- *parallelismThreshold* - It specifies that after how many elements operations in a map are executed in parallel.
- *transformer* - This will transform the data before the data is passed to the specified function.

For example,

```

import java.util.concurrent.ConcurrentHashMap;

class Main {
    public static void main(String[] args) {

        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
    }
}

```

```

        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);

        // forEach() without transformer function
        numbers.forEach(4, (k, v) -> System.out.println("key: " + k +
" value: " + v));

        // forEach() with transformer function
        System.out.print("Values are ");
        numbers.forEach(4, (k, v) -> v, (v) -> System.out.print(v + ",
"));
    }
}

```

## Output

```

ConcurrentHashMap: {One = 1, Two = 2, Three = 3}
key: One value: 1
key: Two value: 2
key: Three value: 3
Values are 1, 2, 3,

```

In the above program, we have used parallel threshold **4**. This means if the map contains 4 entries, the operation will be executed in parallel.

## Variation of forEach() Method

- `forEachEntry()` - executes the specified function for each entry
- `forEachKey()` - executes the specified function for each key
- `forEachValue()` - executes the specified function for each value

## 2. search() Method

The `search()` method searches the map based on the specified function and returns the matched entry.

Here, the specified function determines what entry is to be searched.

It also includes an optional parameter *parallelThreshold*. The parallel threshold specifies that after how many elements in the map the operation is executed in parallel.

For example,

```
import java.util.concurrent.ConcurrentHashMap;

class Main {
    public static void main(String[] args) {

        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);

        // Using search()
        String key = numbers.search(4, (k, v) -> {return v == 3 ? k:
null;});
        System.out.println("Searched value: " + key);

    }
}
```

## Output

```
ConcurrentHashMap: {One=1, Two=2, Three=3}
Searched value: Three
```

## Variants of search() Method

- **searchEntries()** - search function is applied to key/value mappings
- **searchKeys()** - search function is only applied to the keys
- **searchValues()** - search function is only applied to the values

## 3. reduce() Method

The reduce() method accumulates (gather together) each entry in a map. This can be used when we need all the entries to perform a common task, like adding all the values of a map.

It includes two parameters.

- *parallelismThreshold* - It specifies that after how many elements, operations in a map are executed in parallel.
- *transformer* - This will transform the data before the data is passed to the specified function.

For example,

```
import java.util.concurrent.ConcurrentHashMap;

class Main {
    public static void main(String[] args) {

        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);

        // Using search()
        int sum = numbers.reduce(4, (k, v) -> v, (v1, v2) -> v1 + v2);
        System.out.println("Sum of all values: " + sum);

    }
}
```

## Output

```
ConcurrentHashMap: {One=1, Two=2, Three=3}
Sum of all values: 6
```

In the above program, notice the statement

```
numbers.reduce(4, (k, v) -> v, (v1, v2) -> v1+v2);
```

Here,

- 4 is a parallel threshold
- $(k, v) \rightarrow v$  is a transformer function. It transfers the key/value mappings into values only.
- $(v1, v2) \rightarrow v1+v2$  is a reducer function. It gathers together all the values and adds all values.

## Variants of reduce() Method

- **reduceEntries()** - returns the result of gathering all the entries using the specified reducer function
- **reduceKeys()** - returns the result of gathering all the keys using the specified reducer function
- **reduceValues()** - returns the result of gathering all the values using the specified reducer function

## ConcurrentHashMap vs HashMap

Here are some of the differences between ConcurrentHashMap and HashMap,

- ConcurrentHashMap is a **thread-safe** collection. That is, multiple threads can access and modify it at the same time.
- ConcurrentHashMap provides methods for bulk operations like `forEach()`, `search()` and `reduce()`.

### Why ConcurrentHashMap?

- The ConcurrentHashMap class allows multiple threads to access its entries concurrently.
- By default, the concurrent hashmap is divided into **16 segments**. This is the reason why 16 threads are allowed to concurrently modify the map at the same time. However, any number of threads can access the map at a time.
- The `putIfAbsent()` method will not override the entry in the map if the specified key already exists.
- It provides its own synchronization.