**Experiment-2**

**Implement the following Data structures in Java**

**i) Map ii) Sets**

**(i) Introduction to Maps in JAVA**

# Map Interface in Java

The map interface is present in java.util package represents a mapping between a key and a value.

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some common scenarios are as follows:

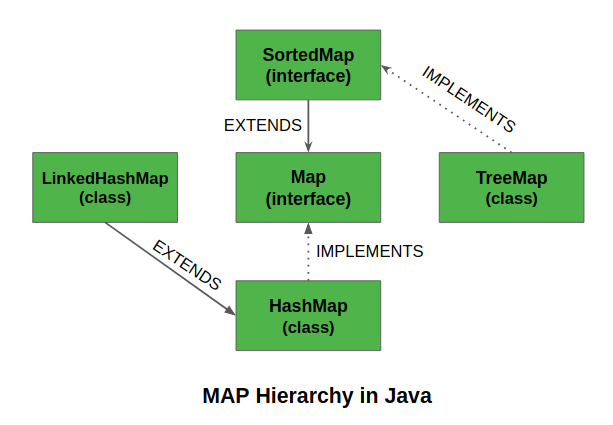
* A map of error codes and their descriptions.
* A map of zip codes and cities.
* A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
* A map of classes and students. Each class (key) is associated with a list of students (value).

Since Map is an interface, objects cannot be created of the type map. We always need a class that extends this map in order to create an object.

### Characteristics of a Map Interface

1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.
2. The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.
3. There are two interfaces for implementing Map in java. They are Map and SortedMap, and three classes: HashMap, TreeMap, and LinkedHashMap.

Classes that implement the Map interface are depicted in the below media and described later as follows:



// Java program to demonstrate the working of Map interface

// Importing required classes

import java.util.\*;

// Main class

class MapDemo {

// Main driver method

public static void main(String args[])

{

// Creating an empty HashMap

Map<String, Integer> hm

= new HashMap<String, Integer>();

// Inserting pairs in above Map

// using put() method

hm.put("a", new Integer(100));

hm.put("b", new Integer(200));

hm.put("c", new Integer(300));

hm.put("d", new Integer(400));

// Traversing through Map using for-each loop

for (Map.Entry<String, Integer> me :

hm.entrySet()) {

// Printing keys

System.out.print(me.getKey() + ":");

System.out.println(me.getValue());

}

}

}

**Output:**

a:100

b:200

c:300

d:400

#### Class 1: **HashMap**

It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. This class uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String. A shorter value helps in indexing and faster searches. Let’s see how to create a map object using this class.

**Example**

// Java Program to illustrate the Hashmap Class

// Importing required classes

import java.util.\*;

// Main class

public class MapDemo {

// Main driver method

public static void main(String[] args)

{

// Creating an empty HashMap

Map<String, Integer> map = new HashMap<>();

// Inserting entries in the Map

// using put() method

map.put("vishal", 10);

map.put("sachin", 30);

map.put("vaibhav", 20);

// Iterating over Map

for (Map.Entry<String, Integer> e : map.entrySet())

// Printing key-value pairs

System.out.println(e.getKey() + " "

+ e.getValue());

}

}

**Output**

vaibhav 20

vishal 10

sachin 30

#### Class 2: **LinkedHashMap**

LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order. Let’s see how to create a map object using this class.

**Example**

// Java Program to Illustrate the LinkedHashmap Class

// Importing required classes

import java.util.\*;

// Main class

public class MapDemo {

// Main driver method

public static void main(String[] args)

{

// Creating an empty LinkedHashMap

Map<String, Integer> map = new LinkedHashMap<>();

// Inserting pair entries in above Map

// using put() method

map.put("vishal", 10);

map.put("sachin", 30);

map.put("vaibhav", 20);

// Iterating over Map

for (Map.Entry<String, Integer> e : map.entrySet())

// Printing key-value pairs

System.out.println(e.getKey() + " "

+ e.getValue());

}

}

**Output:**

vishal 10

sachin 30

vaibhav 20

#### Class 3: **TreeMap**

The TreeMap in Java is used to implement the Map interface and NavigableMap along with the Abstract Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. Let’s see how to create a map object using this class.

// Java Program to Illustrate TreeMap Class

// Importing required classes

import java.util.\*;

// Main class

public class MapDemo {

// Main driver method

public static void main(String[] args)

{

// Creating an empty TreeMap

Map<String, Integer> map = new TreeMap<>();

// Inserting custom elements in the Map

// using put() method

map.put("vishal", 10);

map.put("sachin", 30);

map.put("vaibhav", 20);

// Iterating over Map using for each loop

for (Map.Entry<String, Integer> e : map.entrySet())

// Printing key-value pairs

System.out.println(e.getKey() + " "

+ e.getValue());

}

}

**Output:**

sachin 30

vaibhav 20

vishal 10

**Implementation of Map in JAVA**

A simple map with the option to add, get, remove and get the size of the Map could be implemented like the following. Please note that this map is not very fast for large sets.

import java.util.Arrays;

class MyEntry<K, V> {

private final K key;

private V value;

public MyEntry(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() {

return key;

}

public V getValue() {

return value;

}

public void setValue(V value) {

this.value = value;

}

}

class MyMap<K, V> {

private int size;

private int DEFAULT\_CAPACITY = 16;

@SuppressWarnings("unchecked")

private MyEntry<K, V>[] values = new MyEntry[DEFAULT\_CAPACITY];

public V get(K key) {

for (int i = 0; i < size; i++) {

if (values[i] != null) {

if (values[i].getKey().equals(key)) {

return values[i].getValue();

}

}

}

return null;

}

public void put(K key, V value) {

boolean insert = true;

for (int i = 0; i < size; i++) {

if (values[i].getKey().equals(key)) {

values[i].setValue(value);

insert = false;

}

}

if (insert) {

ensureCapa();

values[size++] = new MyEntry<K, V>(key, value);

}

}

private void ensureCapa() {

if (size == values.length) {

int newSize = values.length \* 2;

values = Arrays.copyOf(values, newSize);

}

}

public int size() {

return size;

}

public void remove(K key) {

for (int i = 0; i < size; i++) {

if (values[i].getKey().equals(key)) {

values[i] = null;

size--;

condenseArray(i);

}

}

}

private void condenseArray(int start) {

for (int i = start; i < size; i++) {

values[i] = values[i + 1];

}

}

}

public class testMap {

public static void main(String args[])

{

// MyMap

MyMap<String, Integer> map = new MyMap<String, Integer>();

map.put("Lars", 1);

map.put("Lars", 2);

map.put("Lars", 11);

System.out.println(map.get("Lars"));

for (int i = 0; i < 100; i++) {

map.put(String.valueOf(i), i);

}

System.out.println(map.size());

System.out.println(map.get("51"));

}

}

**(ii) Introduction to Sets in JAVA**

**Sets in JAVA**

The Java Collection Framework contains numerous interfaces, one of which is the Set Interface. A Set can be defined as a collection of unordered elements; wherein duplicate values cannot be stored. It extends Collection and thus all methods in the Collection interface are available in the Set interface. It is implemented by HashSet, LinkedHashSet, or the TreeSort.

Each of these implementations act differently while iterating the set, mostly with respect to the ordering of the elements, and the time taken for insertion and for accessing the elements.

Hash Set does not provide any guarantee about the order of the elements while iterating the set.

LinkedHashSet on the other hand, provides a guarantee about the order of the elements while iterating them.

TreeSet provides guarantee, but the set is sorted according to the natural order, or by a specific comparator implementation.``

**How to create a Set?**

The following code defines the method of creating a new set:

Set<Integer> num = new HashSet<>();

We have used generics to declare the set of an integer type.

## **Set Methods in Java:**

We can perform multiple operations on a set such as follows😙\*

Add Method

The add method inserts an element to the [Java collection](https://www.edureka.co/blog/java-collections). In the code below, we insert a set of names.

Set<String> strName = new HashSet<>();

strName.add("John");

strName.add("Doe");

System.out.println(strName);

**Output:**

[John,  Doe]

### **Remove Method**

This method removes the specified element from the set.

import java.util.\*;

public class SetsDemo{

public static void main(String args[])

{

// Creating an Empty Set

Set<String> set = new HashSet<String>();

//Adding elements to the set

set.add("John");

set.add("Doe");

// Display the set

System.out.println("Set: " + set);

// Removing the element “Doe” using remove() method

set.remove("Doe");

// Displaying the modified set

System.out.println("Set : "

+ set);

}

}

Output

Set : [John,  Doe]

Set : [John]

### **Is Empty Method**

This method checks determines whether the [set](https://www.edureka.co/blog/java-collections/#sets) is empty is not. It returns true if the set is empty, and false if otherwise.

import java.io.\*;

import java.util.\*;

public class SetsDemo {

public static void main(String args[])

{

Set<String> javaSet = new HashSet<String>();

// Adding elements to the Set

javaSet.add("John");

javaSet.add("Doe");

// Display the set

System.out.println("Set: " + javaSet);

// Checking whether the set is empty

System.out.println("Empty Set : " + javaSet.isEmpty());

// Clearing the set using the clear() method

javaSet.clear();

// Checking whether the set is empty

System.out.println("Empty Set : " + javaSet.isEmpty());

}

}

**Output:**

Set : [John,  Doe]

Empty Set : false

Empty Set : true

### **Size Method**

The size() method returns the size of the set, i.e. the number of elements present in the set.

import java.util.\*;

public class SetsDemo {

public static void main(String args[])

{

// Creating a set

Set<String> set = new HashSet<String>();

set.add("John");

set.add("Doe");

System.out.println("Set: " + set);

// Displaying the size of the sent

System.out.println("Size of the set : " + set.size());

}

}

**Output:**

Set : [John,  Doe]

Size of the set : 2

### **Iterating Over A Set**

We can iterate over all the elements present in the set by the following method:

import java.util.\*;

import java.util.HashSet;

public class SetsDemo {

public static void main(String args[])

{

// Creating a HashSet

HashSet<String> javaSet = new HashSet<String>();

javaSet.add("John");

javaSet.add("Doe");

// Displaying the set

System.out.println("HashSet: " + javaSet);

// Creating an iterator

Iterator itr = javaSet.iterator();

// Displaying the values after iteration

System.out.println("Iterator values: ");

while (itr.hasNext()) {

System.out.println(itr.next());

}

}

}

**Output:**

HashSet : [John,  Doe]

Iterator Values:

John

Doe

**Searching in  A Set**

We use the contains() method to determine whether the set contains a specified element. Returns true if the element is found and false otherwise.

import java.io.\*;

import java.util.HashSet;

public class SetsDemo {

public static void main(String args[])

{

// Creating a HashSet

HashSet<String> javaSet = new HashSet<String>();

javaSet.add("John");

javaSet.add("Doe");

// Displaying the HashSet

System.out.println("HashSet: " + javaSet);

// Checking for “John” in the set

System.out.println("John in set: " + javaSet.contains("John"));

// Checking for "Hazel" in set

System.out.println("Hazel in set: " + javaSet.contains("Hazel"));

}

}

**Output:**

HashSet : [John,  Doe]

John in set: true

Hazel in set: false

## ****Basic Operation on Sets in Java****

**Union:**To add one set to another, we use the Union operation

**Intersection:** To retain the common values from both the sets, we use the intersection operation.

**Difference:**To remove the values of one set, from the other set, the difference operation is used.

import java.util.\*;

public class SetsDemo

{

    public static void main(String args[])

    {

        Set<Integer> d = new HashSet<Integer>();

        d.addAll(Arrays.asList(new Integer[] {3, 2, 1, 9, 6, 4, 0}));

        Set<Integer> e = new HashSet<Integer>();

        e.addAll(Arrays.asList(new Integer[] {3, 1, 9, 5, 2, 0, 7,}));

        // Union Operation

        Set<Integer> union = new HashSet<Integer>(d);

        union.addAll(e);

        System.out.println("Union :" + union);

        // Intersection Operation

        Set<Integer> intersection = new HashSet<Integer>(d);

        intersection.retainAll(e);

        System.out.println("Intersection :" + intersection);

        // Difference Operation

        Set<Integer> difference = new HashSet<Integer>(d);

        difference.removeAll(e);

        System.out.println("Difference :" + difference);

    }

}

**Output:**

Union : [0, 1, 2, 3, 4, 5, 6, 7, 9]

Intersection : [0, 1, 2, 3, 9]

Difference : [4, 6]