Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:

Java Map Hierarchy

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

**import** java.util.\*;

**class** MapExample2{

**public** **static** **void** main(String args[]){

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

  map.put(100,"Amit");

  map.put(101,"Vijay");

  map.put(102,"Rahul");

**for**(Map.Entry m:map.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

}

# Java HashMap

Java **HashMap** class implements the Map interface which allows us to store key and value pair, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

**import** java.util.\*;

**public** **class** HashMapExample1{

**public** **static** **void** main(String args[]){

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

   map.put(1,"Mango");

   map.put(2,"Apple");

   map.put(3,"Banana");

   map.put(4,"Grapes");

   System.out.println("Iterating Hashmap...");

**for**(Map.Entry m : map.entrySet()){

    System.out.println(m.getKey()+" "+m.getValue());

   }

}

}

Java TreeMap class

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order.

**import** java.util.\*;

**class** TreeMap1{

**public** **static** **void** main(String args[]){

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

      map.put(100,"Amit");

      map.put(102,"Ravi");

      map.put(101,"Vijay");

      map.put(103,"Rahul");

**for**(Map.Entry m:map.entrySet()){

       System.out.println(m.getKey()+" "+m.getValue());

      }

 }

}

### **Java TreeMap Example: remove()**

**import** java.util.\*;

**public** **class** TreeMap2 {

**public** **static** **void** main(String args[]) {

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

      map.put(100,"Amit");

      map.put(102,"Ravi");

      map.put(101,"Vijay");

      map.put(103,"Rahul");

      System.out.println("Before invoking remove() method");

**for**(Map.Entry m:map.entrySet())

      {

          System.out.println(m.getKey()+" "+m.getValue());

      }

      map.remove(102);

      System.out.println("After invoking remove() method");

**for**(Map.Entry m:map.entrySet())

      {

          System.out.println(m.getKey()+" "+m.getValue());

      }

      }

}

# Difference between Array and ArrayList

|  |  |  |
| --- | --- | --- |
| **Basis** | **Array** | **ArrayList** |
| **Definition** | An **array** is a dynamically-created object. It serves as a container that holds the constant number of values of the same type. It has a contiguous memory location. | The **ArrayList** is a class of Java **Collections** framework. It contains popular classes like **Vector, HashTable**, and **HashMap**. |
| **Static/ Dynamic** | Array is **static** in size. | ArrayList is **dynamic** in size. |
| **Resizable** | An array is a **fixed-length** data structure. | ArrayList is a **variable-length** data structure. It can be resized itself when needed. |
| **Initialization** | It is mandatory to provide the size of an array while initializing it directly or indirectly. | We can create an instance of ArrayList without specifying its size. Java creates ArrayList of default size. |
| **Performance** | It performs **fast** in comparison to ArrayList because of fixed size. | ArrayList is internally backed by the array in Java. The resize operation in ArrayList slows down the performance. |
| **Primitive/ Generic type** | An array can store both **objects** and **primitives** type. | We cannot store **primitive** type in ArrayList. It automatically converts primitive type to object. |
| **Iterating Values** | We use **for** loop or **for each** loop to iterate over an array. | We use an **iterator** to iterate over ArrayList. |
| **Type-Safety** | We cannot use generics along with array because it is not a convertible type of array. | ArrayList allows us to store only **generic/ type, that's why it is type-safe.** |
| **Length** | Array provides a **length** variable which denotes the length of an array. | ArrayList provides the **size()** method to determine the size of ArrayList. |
| **Adding Elements** | We can add elements in an array by using the **assignment**operator. | Java provides the **add()** method to add elements in the ArrayList. |
| **Single/ Multi-Dimensional** | Array can be **multi-dimensional**. | ArrayList is always **single-dimensional**. |

# Difference between HashSet and HashMap class in Java

|  |  |  |
| --- | --- | --- |
| **Basis** | **HashMap** | **HashSet** |
| **Definition** | Java HashMap is a hash table based implementation of Map interface. | HashSet is a Set. It creates a collection that uses a  hash table for storage. |
| **Implementation** | HashMap implements **Map, Cloneable, and Serializable** interface es. | HashSet implements **Set, Cloneable, Serializable,**  **Iterable** and **Collection** interfaces. |
| **Stores** | In HashMap we store a **key-value pair**. It maintains the mapping of key and value. | In HashSet, we store **objects**. |
| **Duplicate values** | It does not allow **duplicate keys**, but **duplicate values** are **allowed**. | It does not allow **duplicate values**. |
| **Null values** | It can contain a **single null key** and **multiple null values**. | It can contain **a single null value**. |
| **Method of insertion** | HashMap uses the **put()** method to add the elements in the HashMap. | HashSet uses the **add()** method to add elements in the HashSet. |
| **Performance** | HashMap is **faster/ than HashSet because values are associated with a unique key.** | HashSet is **slower** than HashMap because the member object is used for calculating hashcode value, which can be same for two objects. |
| **The Number of objects** | Only **one** object is created during the add operation. | There are **two** objects created during put operation, one for **key** and one for **value**. |
| **Storing Mechanism** | HashMap internally uses **hashing** to store objects. | HashSet internally uses a **HashMap** object to store objects. |
| **Uses** | Always prefer when we do not maintain the **uniqueness**. | It is used when we need to maintain the **uniqueness** of data. |
| **Example** | **{a->4, b->9, c->5}** Where **a, b, c** are **keys** and **4, 9, 5** are **values** associated with key. | **{6, 43, 2, 90, 4}** It denotes a set. |

# Difference between HashMap and TreeMap

|  |  |  |
| --- | --- | --- |
| **Basis** | **HashMap** | **TreeMap** |
| **Definition** | Java **HashMap** is a hashtable based implementation of Map interface. | Java **TreeMap** is a Tree structure-based implementation of Map interface. |
| **Interface Implements** | HashMap implements **Map, Cloneable**, and **Serializable** interface. | TreeMap implements **NavigableMap, Cloneable**, and **Serializable** interface. |
| **Null Keys/ Values** | HashMap allows a **single** null key and **multiple** null values. | TreeMap does not allow **null** keys but can have **multiple** null values. |
| **Homogeneous/ Heterogeneous** | HashMap allows heterogeneous elements because it does not perform sorting on keys. | TreeMap allows homogeneous values as a key because of sorting. |
| **Performance** | HashMap is **faster** than TreeMap because it provides constant-time performance that is O(1) for the basic operations like get() and put(). | TreeMap is **slow** in comparison to HashMap because it provides the performance of O(log(n)) for most operations like add(), remove() and contains(). |
| **Data Structure** | The HashMap class uses the **hash table**. | TreeMap internally uses a **Red-Black** tree, which is a self-balancing Binary Search Tree. |
| **Comparison Method** | It uses **equals()** method of the **Object** class to compare keys. The equals() method of Map class overrides it. | It uses the **compareTo()** method to compare keys. |
| **Functionality** | HashMap class contains only basic functions like **get(), put(), KeySet()**, etc. . | TreeMap class is rich in functionality, because it contains functions like: **tailMap(), firstKey(), lastKey(), pollFirstEntry(), pollLastEntry()**. |
| **Order of elements** | HashMap does not maintain any order. | The elements are sorted in **natural order** (ascending). |
| **Uses** | The HashMap should be used when we do not require key-value pair in sorted order. | The TreeMap should be used when we require key-value pair in sorted (ascending) order. |