How to Learn Java Collections – A Complete Guide

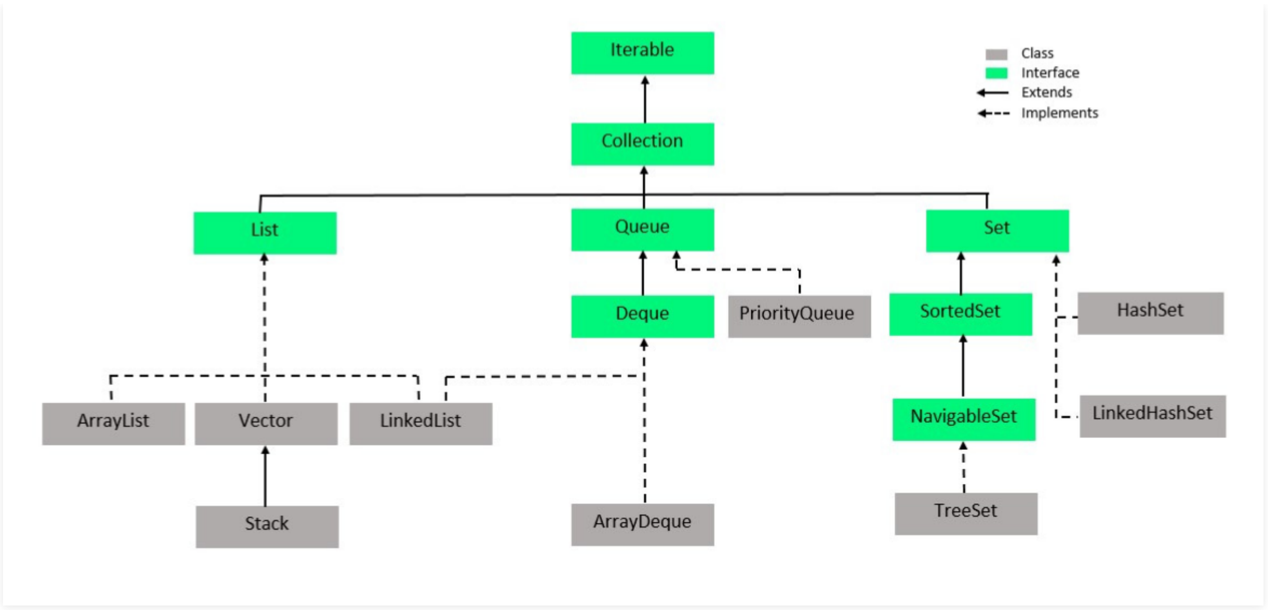
Difficulty Level : Medium Last Updated : 29 Nov, 2021

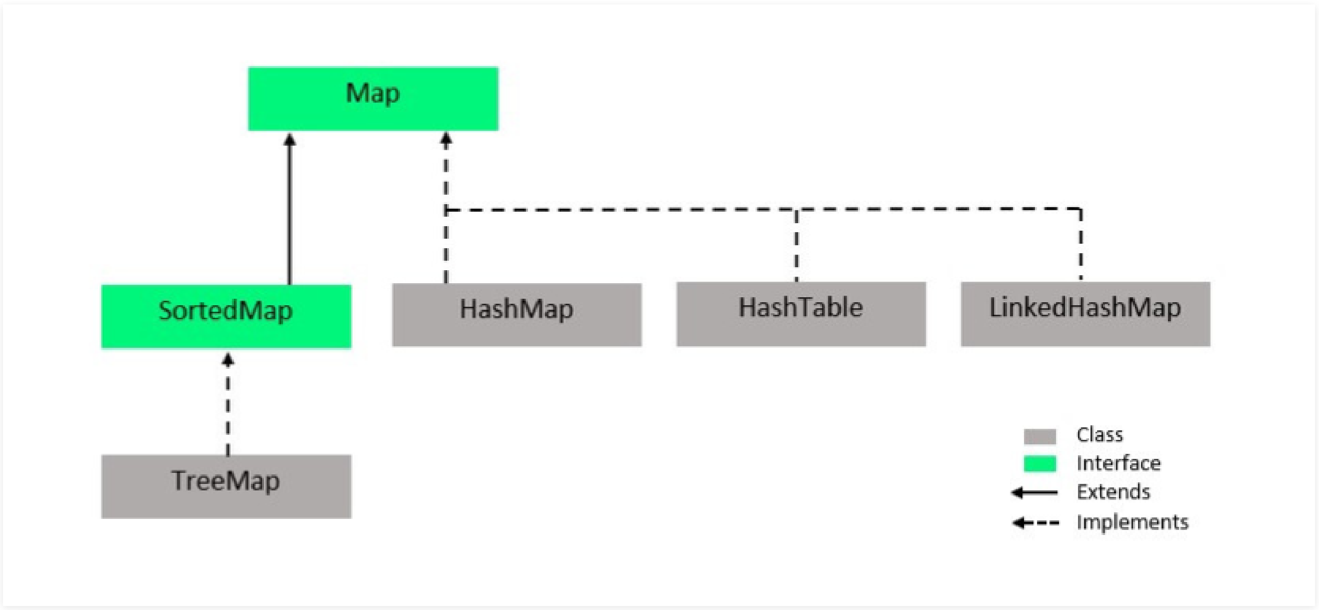
In the real world, a collection by definition is a group of articles that have similar properties and attributes. Since Java is an Object-Oriented Language it mimics the real world. In Java, a **Collection** is a group of multiple objects put together into a single unit. Java Collections is a very vast topic and as a beginner can be difficult to navigate your way while learning it. Here we have everything you need to know while starting off with Java Collections.

What is a Collection Framework?

We have our collection of objects, now we need an organized way to use these collections, therefore we need a framework. The Java Collection Framework, first introduced in JDK 1.2 (Java Development Kit 1.2), is an architecture made up of interfaces and classes. In simple words, it is like a skeletal structure for components that is ready to use for various programming needs. It also offers different data operations like searching, sorting, insertion, deletion, and manipulation. All of the classes and interfaces of the collection framework are bundled into the java.util package.

The Collection Framework Hierarchy





Class Vs Interface

Class	Interface
A class is a user-defined prototype for building objects in Java.	An interface is a user-defined blueprint that describes the structure of every class that implements it.
It is used to define objects.	It cannot be used to define objects.
A class can have access modifiers public and default.	An Interface can have access modifiers public and default.
Classes can be concrete or abstract.	All interfaces are abstract.

Class	Interface
A class consists of constructors, methods and attributes. The methods are defined in a class.	An interface consists attributes and methods. The methods are not defined in an interface, it only contains their signature.

Now that we have the basic concepts of what Java collections are made of we will understand each of its components in detail. What are their properties and a few examples of the most used collections?

1. Iterable

The Iterable interface is the root of the entire collection hierarchy, which means that every class and interface implements it. The primary function of an iterator is to allow the user to traverse through all of the collection class objects as if they were simple sequences of data items.

2. Collection

The Collection interface extends the Iterable interface. It has the basic methods required for using all the other collections in the framework to add, delete, and manipulate data. Since it is an interface it only has a method signature (i.e. <return type> methodName (ArgumentList);) and no definition because every interface or class that implements this interface will have different types of elements to handle. But since they implement this interface there is uniformity and structure to the rest of the collection. The methods of the collection interface are given below, all the interfaces and classes that extend or implement the Collection interface use these methods along with their own added methods specific to them.

3. List

The List interface extends from the Collection interface. The elements in a list are ordered like a sequence. The user can use the index number to access a particular element in the list, that is to say, the user has complete control over which element is inserted wherein the list.

3. a) ArrayList

The ArrayList_class implements the List interface. The objects of this class are dynamic arrays. The ArrayList is essentially a resizable implementation of List. It implements all of the List methods and allows all elements even null elements. The ArrayList objects have a capacity, which is initially equal to the size but increases dynamically as new elements are added. An ArrayList is unsynchronised, which means multiple threads can access them at the same time. A thread is a unit of sequential flow control that can be processed in the Operating System.

Syntax:

```
ArrayList<?> arrayListName = new ArrayList<?>();
```

Example: Now we will take an example and perform some basic operations on an ArrayList. Here we instantiate an ArrayList named intArr. We use the add() method to add integers to intArr. The Integer class used in declaring intArr is a wrapper class for that basic datatype int. Wrapper classes extend from the Object class and they are used so that basic datatypes are compatible with other classes. Next, we print the ArrayList on the console. We use the remove() method to remove elements from the specified indices. We check if an element, 25 here, exists in intArr and print the appropriate message. Then we retrieve the element at index 1 using the get() method. As you can observe when an element is removed using remove() method the rest of the elements shift in sequence.

```
// An example for ArrayList
// All of the classes and
// interfaces of the collection
// framework are bundled into
// the java.util package
import java.util.*;

public class BasicArrayList {

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

        // Instantiate an ArrayList Object
        // Integer is a wrapper class for
        // the basic datatype int
        ArrayList<Integer> intArr = new ArrayList<Integer>();

        // Add elements using add() method
        intArr.add(10);
        intArr.add(12);
        intArr.add(25);
        intArr.add(19);
        intArr.add(11);
        intArr.add(3);

        // Print the ArrayList on the console
        System.out.println(intArr);

        // Remove elements at index 1 and 4
        intArr.remove(1);
        intArr.remove(4);

        // Print the ArrayList on the console
        System.out.println(intArr);

        // Check if intArr contains the element 25
        if(intArr.contains(25))
        {
            System.out.println("The ArrayList contains 25");
        }
        else
        {
            System.out.println("No such element exists");
        }

        // Use get method to get the element at index 1
        int elementAt1 = intArr.get(1);
        System.out.println("The Element at index 1 now is " + elementAt1);

    }

}
```

Output

```
[10, 12, 25, 19, 11, 3]
[10, 25, 19, 11]
The ArrayList contains 25
The Element at index 1 now is 25
```

3. b) Vector

The vector class implements the List iterator. A Vector instance is a dynamic array, wherein the elements can be accessed with indices. The difference between a Vector an ArrayList is that Vectors are synchronized.

Syntax:

```
Vector<?> vectorName = new Vector<?>();
```

Let us better understand Vector with an example, In the code given below we have declared a Vector named intVector, we use add() to add elements to the Vector. The size() method gives the current number of elements stored in the Vector. The remove() method is used to remove an element at the specified index.

```
// An example for Vector
import java.util.*;

public class VectorExample {

    public static void main(String[] args) {

        // Instantiate Vector object
        Vector<Integer> intVector = new Vector<Integer>();

        // Print the initial size of the Vector
        System.out.println("The initial size of the Vector = " + intVector.size
        System.out.println();

        // Add elements using add method
        intVector.add(11);
        intVector.add(18);
        intVector.add(1);
        intVector.add(87);
        intVector.add(19);
        intVector.add(11);

        // Print the Vector on the console
        System.out.println("The Vector intVector : ");
        System.out.println(intVector);
        System.out.println("Size of intVector : " + intVector.size());

        System.out.println();

        // Remove the element at index 2
        intVector.remove(2);
        // Print the vector again on the console
        System.out.println("The Vector intVector after removing element at 2 :
        System.out.println(intVector);

        System.out.println();

        // Clear all elements of the Vector and
        // Print the Vector on the console
        intVector.clear();
        System.out.println("The Vector intVector after using clear : ");
        System.out.println(intVector);

    }

}
```

Output

```
The initial size of the Vector = 0

The Vector intVector :
[11, 18, 1, 87, 19, 11]
Size of intVector : 6

The Vector intVector after removing element at 2 :
[11, 18, 87, 19, 11]

The Vector intVector after using clear :
[]
```

3. b) i) Stack

The stack class extends from the Vector class. The Stack is a last-in-first-out (LIFO) structure. You can visualize it as a stack of books on a table the book that is kept first has to e retrieved last, and the book that is kept on the stack last has to be retrieved first. The basic methods of the stack class are push, pop, peek, empty, and search.

Syntax:

```
Stack<?> stackName = new Stack<?>();
```

Example: Let us understand Stack better with an example. In the code given below, we first instantiate a Stack named strStack, whose elements are of type String. The elements are added using the push() method. The size() method returns the number of elements present in the Stack. The search() method is used to search an element in the stack. It returns the 1-based position of the element if found else -1 is returned to indicate no such element exists in the Stack.

```
// An example to show workings of a Stack
import java.util.*;

public class StackExample {

    public static void main(String[] args) {

        // Instantiate a Stack named strStack
        Stack<String> strStack = new Stack<String>();

        // Add elements using the push() method
        strStack.push("Stack");
        strStack.push("a");
        strStack.push("is");
        strStack.push("This");

        // The size() method gives the
        // number of elements in the Stack
        System.out.println("The size of the Stack is : " + strStack.size());

        // The search() method is
        // used to search an element
        // it returns the position of
        // the element
        int position = strStack.search("a");
        System.out.println("\nThe string 'a' is at position " + position);

        System.out.println("\nThe elements of the stack are : ");
        String temp;
        int num = strStack.size();

        for(int i = 1; i <= num; i++)
        {
            // peek() returns the topmost element
            temp = strStack.peek();
            System.out.print(temp + " ");

            // pop() removes the topmost element
            strStack.pop();
            }

        }

    }
```

Output

```
The size of the Stack is : 4

The string 'a' is at position 3

The elements of the stack are :
This is a Stack
```

3. c) LinkedList

The LinkedList class implements the List interface as well as the Deque interface. The LinkedList is the class implementation of the linked list data structure, where every element has a pointer to the next element forming a link. Since each element has an address of the next element, the linked list elements, referred to as nodes, can be stored at non-contiguous locations in memory.

Syntax:

```
LinkedList<?> linkedListName = new LinkedList<?>();
```

Let us take an example to understand LinkedList. In the code given below, we instantiate a LinkedList named strLinkedList. The add() method is used to add elements and remove() method to remove elements. The retrieval of the elements is done using get() method.

```
// An example for the LinkedList
import java.util.*;
```

```
public class LinkedListExample {

    public static void main(String[] args) {

        // Instantiate LinkedList named strLinkedList
        LinkedList<String> strLinkedList = new LinkedList<String>();

        // Add elements to the LinkedList using add()
        strLinkedList.add("This");
        strLinkedList.add("is");
        strLinkedList.add("a");
        strLinkedList.add("LinkedList");

        // The elements are retrieved using the get() method
        System.out.println("The contents of strLinkedList : ");
        for(int i = 0; i < strLinkedList.size(); i++)
        {
            System.out.print(strLinkedList.get(i) + " ");
        }

        // The elements are removed using remove()
        strLinkedList.remove(0);
        strLinkedList.remove(1);

        System.out.println("\n\nThe contents of strLinkedList after remove oper
        for(int i = 0; i < strLinkedList.size(); i++)
        {
            System.out.print(strLinkedList.get(i) + " ");
        }

    }

}
```

Output

```
The contents of strLinkedList :
This is a LinkedList

The contents of strLinkedList after remove operation :
is LinkedList
```

4. Queue

The Queue interface extends the Collection interface. Queue is the interface implementation of the queue data structure. Since Queue in java is an interface it doesn't have a definition of the methods only their signatures. Queue is typically a first-in-first-out (FIFO) structure, though that is not the case for PriorityQueue. You may visualize it as a queue of people at a counter, the person who enters first gets services first and leaves first.

4. a) PriorityQueue

The PriorityQueue class implements the Queue interface. The elements of a PriorityQueue are either ordered in natural order or in an order specified by a Comparator, which depends on the constructor used. The PriorityQueue is unbounded but there is a capacity that dictates the size of the array in which the elements are stored. The initial capacity is equal to the size of the array but as new elements are added it expands dynamically.

Syntax:

```
PriorityQueue<?> priorityQueueName = new PriorityQueue<?>();
```

Let us understand PriorityQueue better with an example. In the code given below we instantiate an object of PriorityQueue named intPriorityQueue, since no Comparator is specified in the constructor the elements of this PriorityQueue will be naturally ordered. The add() method is used to add elements and remove() is used to remove a single instance of the specified element. The peek() method is implemented from the Queue interface and it returns the element at the head of the PriorityQueue. The poll() method however removes the element at the head of the PriorityQueue and returns it.

```
// An example for PriorityQueue
import java.util.*;

public class PriorityQueueExample {

    public static void main(String[] args) {

        // Instantiate PriorityQueue object named intPriorityQueue
        PriorityQueue<Integer> intPriorityQueue = new PriorityQueue<Integer>();

        // Add elements using add()
        intPriorityQueue.add(17);
        intPriorityQueue.add(20);
        intPriorityQueue.add(1);
        intPriorityQueue.add(13);
        intPriorityQueue.add(87);

        // Print the contents of PriorityQueue
        System.out.println("The contents of intPriorityQueue : ");
        System.out.println(intPriorityQueue);

        // The peek() method is used to retrieve
          // the head of the PriorityQueue
        System.out.println("\nThe head of the PriorityQueue : " + intPriorityQu

        // The remove() method is used
        // to remove a single instance
        // of the specified object
        intPriorityQueue.remove(17);

        // Print the contents of PriorityQueue
        System.out.println("\nThe contents of intPriorityQueue after removing 1
        System.out.println(intPriorityQueue);

        // The poll() method is used
        // to retrieve and remove the
        // element at the head of the PriorityQueue
        Integer head = intPriorityQueue.poll();
        System.out.println("\nThe head of the PriorityQueue was : " + head);

        // Print the contents of PriorityQueue
        System.out.println("\nThe contents of intPriorityQueue after poll : ");
        System.out.println(intPriorityQueue);
    }

}
```

Output

```
The contents of intPriorityQueue :
[1, 13, 17, 20, 87]

The head of the PriorityQueue : 1

The contents of intPriorityQueue after removing 17 :
[1, 13, 87, 20]

The head of the PriorityQueue was : 1
The contents of intPriorityQueue after poll :
[13, 20, 87]
```

5. Deque

The Deque interface extends the Queue interface. The Deque is an implementation of the double-ended queue data structure, which is a linear structure where insertion and deletion can be done at both ends of the queue. The Deque interface supports deques that have capacity restrictions, as well as that, have no fixed limit. Deque can be used as a last-in-first-out (LIFO) as well as a first-in-first-out (FIFO) structure.

5. a) ArrayDeque

The ArrayDeque class implements the Deque interface. ArrayDeque is a re-sizable implementation of Deque, it has no fixed capacity but increases as required. The ArrayDeque can be used as a stack, and it is faster compared to the Stack class. ArrayDeque is not thread-safe and it does not allow concurrent access by different threads.

Syntax:

```
ArrayDeque<?> arrayDequeName = new ArrayDeque<?>();
```

6. Set

The Set_interface extends the Collection interface. The Set is a structure that models the mathematical definition of a set. It is a collection of objects and no duplicate objects are allowed. The Set allows at most one null element.

6. a) HashSet

The HashSet_class implements the Set interface. In a HashSet, the order of the elements may not be the same as the order of insertion. When an element is added into the HashSet a HashCode is calculated and the element is added to the appropriate bucket (a bucket is a slot in any Hash structure). A good HashSet algorithm will uniformly distribute the elements so that the time performance of the structure remains constant. A constant-time performance means it takes constant time for basic operations like insert, deletes, and search.

6. b) LinkedHashSet

LinkedHashSet implements the Set interface. The LinkedHashSet is very similar to the HashSet with the difference being that for every bucket the structure uses to store elements is a doubly-linked list. The LinkedHashSet ordering is better compared to HashSet without any additional costs.

Let us take an example to understand HashSet and LinkedHashSet. In the code given below, we instantiate a HashSet named strHashSet, using add() method add elements to the HashSet. The hasNext() method and next() method are methods of the Iterable interface that are used to check if there is the next element and to retrieve the next element respectively in any Collection. Using the constructor of the LinkedHashSet, all of the elements of the HashSet are added to it. An Iterator is created to traverse through it and using it the elements are printed on the console.

```java
// An example for HashSet and LinkedHashSet
import java.util.*;

public class HashSetAndLinkedHashSet {

    public static void main(String[] args) {

        /*-----------HashSet-------------*/

        // Instantiate a HashSet object named strHashSet
        HashSet<String> strHashSet = new HashSet<String>();

        // Add elements using add()
        strHashSet.add("This");
        strHashSet.add("is");
        strHashSet.add("a");
        strHashSet.add("HashSet");

        // Create an Iterator to traverse through the HashSet
        Iterator<String> hsIterator = strHashSet.iterator();

        // Print all the elements of the HashSet
        System.out.println("Contents of HashSet : ");
        while(hsIterator.hasNext())
        {
            System.out.print(hsIterator.next() + " ");
        }

        /*---------LinkedHashSet----------*/

        // Instantiate an object of LinkedHashSet named strLinkedHashSet
        // Pass the name of the HashSet created earlier to copy all of the cont
        // of the HashSet to the LinkedHashSet using a constructor
        LinkedHashSet<String> strLinkedHashSet = new LinkedHashSet<String>(strH

        // Create an Iterator to traverse through the LinkedHashSet
        Iterator<String> lhsIterator = strLinkedHashSet.iterator();

        // Print all the elements of the LinkedHashSet
        System.out.println("\n\nContents of LinkedHashSet : ");
        while(lhsIterator.hasNext())
        {
            System.out.print(lhsIterator.next() + " ");
        }

    }

}
```

Output

```
Contents of HashSet :
a This is HashSet

Contents of LinkedHashSet :
a This is HashSet
```

7. SortedSet

The SortedSet interface extends the Set interface. The SortedSet provides a complete ordering of the elements. The default ordering is by Natural order else it is ordered by a Comparator specified at the time of construction. The traversing typically is in ascending order of elements.

8. NavigableSet

The NavigableSet interface extends from the SortedSet interface. In addition to the methods of the SortedSet, NavigableSet has navigation methods that give closest matches such as floor, ceiling, lower and higher. A NavigableSet can be traversed in ascending and descending order. Although it allows null element implementations it is discouraged as these implementations can give ambiguous results.

8. a) TreeSet

The TreeSet class implements the Navigable interface. The TreeSet as the name suggests uses a tree structure to store elements and a set to order the elements. The ordering is either natural ordering or ordering by the Comparator specified at the time construction. The TreeSet is unsynchronized, which is if multiple threads want to access it at the same time we need to synchronize it externally.

Syntax:

```
TreeSet<?> treeSetName = new TreeSet<?>();
```

Let us take an example to understand TreeSet better. In the code given below, we instantiate an object named intTreeSet. As you can observe the order in this TreeSet is natural ordering and no duplicate elements are allowed. The add() method is used to add the elements and remove() method is used to delete elements.

```
// An example for TreeSet
import java.util.*;

public class TreeSetExample {

    public static void main(String[] args) {

        // Instantiate an object of TreeSet named intTreeSet
        TreeSet<Integer> intTreeSet = new TreeSet<Integer>();

        // Add elements using add()
        intTreeSet.add(18);
        intTreeSet.add(13);
        intTreeSet.add(29);
        intTreeSet.add(56);
        intTreeSet.add(73);
        // Try to add a duplicate
        // Observe output as it will not be added
        intTreeSet.add(18);

        // Print the TreeSet on the console
        System.out.println("The contents of intTreeSet : ");
        System.out.println(intTreeSet);

        // Remove 18 using remove()
        if(intTreeSet.remove(18))
        {
            System.out.println("\nElement 18 has been removed");
        }
        else
        {
            System.out.println("\nNo such element exists");
        }

        // Try to remove a non-existent element
        if(intTreeSet.remove(12))
        {
            System.out.println("\nElement 18 has been removed");
        }
        else
        {
            System.out.println("\nNo such element exists");
        }

        System.out.println();

        // Print the TreeSet on the console
        System.out.println("The contents of intTreeSet : ");
        System.out.println(intTreeSet);

    }

}
```

Output

```
The contents of intTreeSet :
[13, 18, 29, 56, 73]


Element 18 has been removed


No such element exists
```

```
The contents of intTreeSet :
[13, 29, 56, 73]
```

9. Map

The Map interface is a structure that maps a key to every value. A Map does not allow duplicate elements as one key cannot have multiple mappings. A Map has three different views, a Set view of the keys, a Set view of key-value mappings, and a Collection view of the values. The methods of the Map interface are given below, every class that implements Map must provide definitions for these methods.

9. a) HashMap

The HashMap_class implements the Map interface. For every entry in a HashMap, a hashCode is computed and this entry is inserted into the bucket with the hashCode value as its index. Every entry is a key-value pair. A bucket in a HashMap may contain more than one entry. A good HashMap algorithm will try to uniformly distribute the elements in the HashMap. HashMap has constant time performance for basic retrieval, insertion, deletion, and manipulation operations. The two most important factors that affect the performance of a HashMap are initial capacity and load factor. The number of buckets is the capacity and the measure of when to increase this capacity is load factor. The HashMap is faster compared to a HashTable.

Syntax:

```
HashMap<? , ?> hashMapName = new HashMap<? , ?>();
```

9. b) Hashtable

The Hashtable_class implements the Map interface. The Hashtable has key-value pairs as its elements. For effective implementation of the hashtable, the keys must be unique. A Hashtable is very similar to a Hashtable, but Hashtable is synchronous. A good Hashtable algorithm will try to uniformly distribute the elements in the Hashtable. Hashtable has constant time performance for basic retrieval, insertion, deletion, and manipulation operations. The two most important factors that affect the performance of a Hashtable are initial capacity and load factor. The number of buckets is the capacity and the measure of when to increase this capacity is load factor.

Syntax:

```
HashTable<? , ?> hashTableName = new HashTable<? , ?>();
```

9. c) LinkedHashMap

The LinkedHashMap class implements the Map interface. A LinkedHashMap is a hash map linked list implementation of a map. Every entry in the LinkedHashMap has a doubly-linked list running through it. This linked list defines the iteration order that is the order of the keys inserted into the LinkedHashMap. Like all implementations of Map, the elements of the LinkedHashMap are key-value pairs.

Syntax:

```
LinkedHashMap<? , ?> linkedHashMapName = new LinkedHashMap<? , ?>();
```

Let us take an example to understand all of the Map implementations. In the code given below, we add elements using put() method to all of the HashMap, Hashtable, and LinkedHashMap. As put() is a method of the Map interface, therefore, is implemented by all three of these classes. As you can observe the insertion order of Hashtable is not the same as the internal ordering, therefore it is nondeterministic. When we try to insert a duplicate key the old value is replaced in all three. And When we try to insert a duplicate value with a different key it is added as a new entry. Basically this is to depict that we can have duplicate values but not duplicate keys.

```
// An example for HashMap,
// Hashtable and LinkedHashMap
import java.util.*;

public class MapImplementaionExample {

    public static void main(String[] args) {

        /*--------------HashMap---------------*/

        // Instantiate an object of HashMap named hashMap
        HashMap<Integer, String> hashMap = new HashMap<Integer, String>();

        // Add elements using put()
        hashMap.put(1, "This");
        hashMap.put(2, "is");
        hashMap.put(3, "HashMap");

        // Print the HashMap contents on the console
        System.out.println("Contents of hashMap : ");
        System.out.print(hashMap.entrySet());

        // Add a duplicate key
        hashMap.put(3, "Duplicate");

        // Add a duplicate value
        hashMap.put(4, "This");

        // Print the HashMap contents on the console
        System.out.println("\nContents of hashMap after adding duplicate : ");
        System.out.print(hashMap.entrySet());

        /*--------------Hashtable----------------*/

        // Instantiate an object of Hashtable named hashTable
        Hashtable<Integer, String> hashTable = new Hashtable<Integer, String>()

        // Add elements using put()
        hashTable.put(11, "This");
        hashTable.put(12, "is");
        hashTable.put(13, "Hashtable");

        // Print the Hashtable contents on the console
        System.out.println("\n\nContents of hashTable : ");
        System.out.print(hashTable.entrySet());

        // Add a duplicate key
        hashTable.put(11, "Duplicate");

        // Add a duplicate value
        hashTable.put(14, "is");

        // Print the Hashtable contents on the console
        System.out.println("\nContents of hashTable after adding duplicate : ")
        System.out.print(hashTable.entrySet());

        /*---------------LinkedHashMap---------------*/

        // Instantiate an object of LinkedHashMap named linkedHashMape
        LinkedHashMap<Integer, String> linkedHashMap = new LinkedHashMap<Intege

        // Add elements using put()
        linkedHashMap.put(21, "This");
        linkedHashMap.put(22, "is");
        linkedHashMap.put(23, "LinkedHashMap");

        // Print the LinkedHashMap contents on the console
        System.out.println("\n\nContents of linkedHashMap : ");
        System.out.print(linkedHashMap.entrySet());
        // Add a duplicate key
        linkedHashMap.put(22, "Duplicate");

        // Add a duplicate value
        linkedHashMap.put(24, "This");

        // Print the LinkedHashMap contents on the console
        System.out.println("\nContents of linkedHashMap after adding duplicate
        System.out.print(linkedHashMap.entrySet());
    }

}
```

Output

```
Contents of hashMap :
[1=This, 2=is, 3=HashMap]
Contents of hashMap after adding duplicate :
[1=This, 2=is, 3=Duplicate, 4=This]

Contents of hashTable :
[13=Hashtable, 12=is, 11=This]
Contents of hashTable after adding duplicate :
[14=is, 13=Hashtable, 12=is, 11=Duplicate]

Contents of linkedHashMap :
[21=This, 22=is, 23=LinkedHashMap]
Contents of linkedHashMap after adding duplicate :
[21=This, 22=Duplicate, 23=LinkedHashMap, 24=This]
```

10. SortedMap

The SortedMap interface extends the Map interface with an added stipulation of a total order of keys. The keys are either ordered by natural ordering or by a Comparator specified at the time of construction, depending on the constructor used. All of the keys must be comparable.

10. a) TreeMap

The TreeMap class implements the SortedMap interface. The TreeMap class uses a red-black tree structure for storage and a map for ordering the elements. Every element is a key-value pair. This implementation gives a guaranteed log(n) time cost for basic operations.

Syntax:

```
TreeMap<? , ?> treeMapName = new TreeMap<? , ?>();
```

Let us take an example to understand the basics of TreeMap. In the code given below, we instantiate an object of the TreeMap object named treeMap. Add elements using put() method. When we try to add a duplicate key with a different value the older instance is replaced with the new value associated with this key. But when we try to add a duplicate value with a new key it is taken as a different entry.

```java
// An example of TreeMap
import java.util.*;

public class TreeMapExample {

    public static void main(String[] args) {

        // Instantiate an object of TreeMap named treeMap
        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();

        // Add elements using put()
        treeMap.put(1, "This");
        treeMap.put(2, "is");
        treeMap.put(3, "TreeMap");

        // Print the contents of treeMap on the console
        System.out.println("The contents of treeMap : ");
        System.out.println(treeMap);

        // Add a duplicate key
        treeMap.put(1, "Duplicate");

        // Add a duplicate value
        treeMap.put(4, "is");

        // Print the contents of treeMap on the console
        System.out.println("\nThe contents of treeMap after adding duplicates :
        System.out.println(treeMap);
    }

}
```

Output

```
The contents of treeMap :
{1=This, 2=is, 3=TreeMap}

The contents of treeMap after adding duplicates :
{1=Duplicate, 2=is, 3=TreeMap, 4=is}
```

GeeksforGeeks has also prepared a course **Fundamentals of Java and Java Collections** to teach you the Java concepts in depth. This course will help you to use Collections Framework's inbuilt classes and functions in order to implement some of the complex data structures easily & efficiently and perform operations on them. Check out this course: