| **ARRAYS** | **COLLECTION** |
| --- | --- |
| Arrays are fixed in size that is once we create an array we can not increased or decreased based on our requirement. | Collection are growable in nature that is based on our requirement. We can increase or decrease of size. |
| With respect to memory Arrays are not recommended to use. | With respect to memory collection are recommended to use. |
|  |  |
| With respect to performance Arrays are recommended to use. | With respect to performance collection are not recommended to use. |
| Arrays can hold only homogeneous data types elements. | Collection can hold both homogeneous and and heterogeneous elements. |
| There is no underlying data structure for arrays and hence redimat method support is not available. | Every collection class is implemented based on some std data structure and hence for every requirement redimat method support is available being a performance. we can use these method directly and We are not responsible to implement these methods. |
| Arrays can hold both object and primitive. | Collection can hold only object types but primitive. |

Collections in Java

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

**Need for Collection Framework :**  
Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members:

|  |
| --- |
| // Java program to show whey collection framework was needed  import java.io.\*;  import java.util.\*;    class TestCollection  {      public static void main (String[] args)      {          // Creating instances of array, vector and hashtable          int arr[] = new int[] {1, 2, 3, 4};          Vector<Integer> v = new Vector();          Hashtable<Integer, String> h = new Hashtable();          v.addElement(1);          v.addElement(2);          h.put(1,"Smita");          h.put(2,"4Smita");            // Array instance creation requires [], while Vector          // and hastable require ()          // Vector element insertion requires addElement(), but          // hashtable element insertion requires put()            // Accessing first element of array, vector and hashtable          System.out.println(arr[0]);          System.out.println(v.elementAt(0));          System.out.println(h.get(1));            // Array elements are accessed using [], vector elements          // using elementAt() and hashtable elements using get()      }  } |

Copy CodeRun on IDE

Output:

1

1

geek

As we can see, none of these collections (Array, Vector or Hashtable) implement a standard member access interface. It was very difficult for programmers to write algorithms that can work for all kinds of Collections. Another drawback being that most of the ‘Vector’ methods are final, meaning we cannot extend the ’Vector’ class to implement a similar kind of Collection.  
***Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced the Collection Framework in JDK 1.2***.

Both legacy Vectors and Hashtables were modified to conform to the Collection Framework.

**Advantages of Collection Framework:**

1. Consistent API : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
2. Reduces programming effort: A programmer doesn’t have to worry about the design of Collection, and he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

**Hierarchy of Collection Framework**

Collection Map

/ / \ \ |

/ / \ \ |

Set List Queue Dequeue SortedMap

/

/

SortedSet

**Core Interfaces in Collections**

Note that this diagram only shows core interfaces.

**Collection :** Root interface with basic methods like add(), remove(),

contains(), isEmpty(), addAll(), ... etc.

**Set :** Doesn't allow duplicates. Example implementations of Set

interface are HashSet (Hashing based) and TreeSet (balanced

BST based). Note that TreeSet implements **SortedSet**.

**List :** Can contain duplicates and elements are ordered. Example

implementations are LinkedList (linked list based) and

ArrayList (dynamic array based)

**Queue :** Typically order elements in FIFO order except exceptions

like PriorityQueue.

**Deque :** Elements can be inserted and removed at both ends. Allows

both LIFO and FIFO.

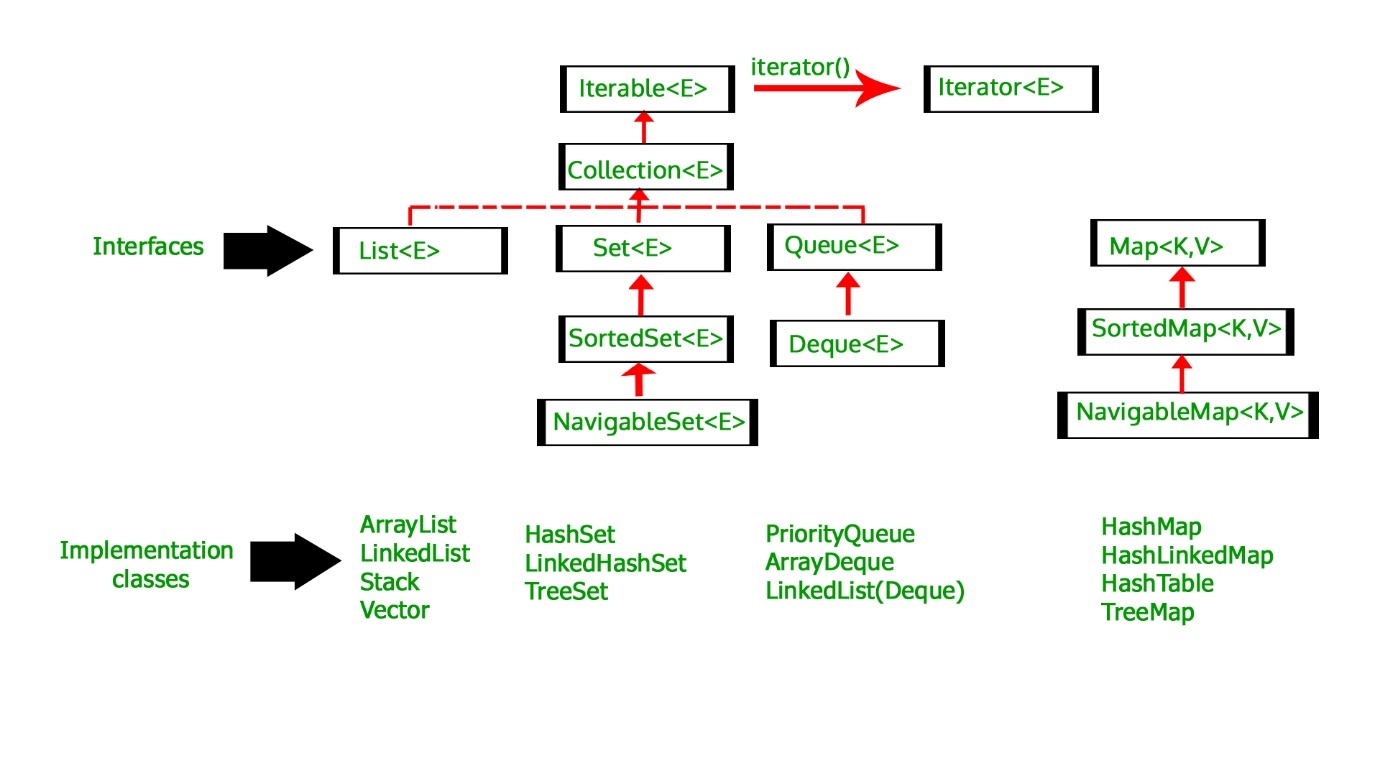
**Map :** Contains Key value pairs. Doesn't allow duplicates. Example

implementation are HashMap and TreeMap.

TreeMap implements **SortedMap**.

The difference between Set and Map interface is that in Set we

have only keys, whereas in Map, we have key, value pairs.



ArrayList in Java

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

* ArrayList inherits AbstractList class and implements List interface.
* ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection.
* Java ArrayList allows us to randomly access the list.
* ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.

**Methods in Java ArrayList:**

1. void clear(): This method is used to remove all the elements from any list.
2. void add(int index, Object element): This method is used to insert a specific element at a specific position index in a list.
3. void trimToSize(): This method is used to trim the capacity of the instance of the ArrayList to the list’s current size.
4. int indexOf(Object O): The index the first occurrence of a specific element is either returned, or -1 in case the element is not in the list.
5. int lastIndexOf(Object O): The index the last occurrence of a specific element is either returned, or -1 in case the element is not in the list.
6. Object clone(): This method is used to return a shallow copy of an ArrayList.
7. Object[] toArray(): This method is used to return an array containing all of the elements in the list in correct order.
8. Object[] toArray(Object[] O): It is also used to return an array containing all of the elements in this list in the correct order same as the previous method.
9. boolean addAll(Collection C): This method is used to append all the elements from a specific collection to the end of the mentioned list, in such a order that the values are returned by the specified collection’s iterator.
10. boolean add(Object o): This method is used to append a specificd element to the end of a list.
11. boolean addAll(int index, Collection C): Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list.

LinkedList in Java

Linked List are linear data structures where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. It also has few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach to a node we wish to access.

To store the elements in a linked list we use a doubly linked list which provides a linear data structure and also used to inherit an abstract class and implement list and deque interfaces.

In Java, LinkedList class implements the list interface. The LinkedList class also consists of various constructors and methods like other java collections.

**Constructors for Java LinkedList:**

1. LinkedList(): Used to create an empty linked list.
2. LinkedList(Collection C): Used to create a ordered list which contains all the elements of a specified collection, as returned by the collection’s iterator.

// Java code for Linked List implementation

import java.util.\*;

public class TestLinkedList

{

public static void main(String args[])

{

// Creating object of class linked list

LinkedList<String> object = new LinkedList<String>();

// Adding elements to the linked list

object.add("A");

object.add("B");

object.addLast("C");

object.addFirst("D");

object.add(2, "E");

object.add("F");

object.add("G");

System.out.println("Linked list : " + object);

// Removing elements from the linked list

object.remove("B");

object.remove(3);

object.removeFirst();

object.removeLast();

System.out.println("Linked list after deletion: " + object);

// Finding elements in the linked list

boolean status = object.contains("E");

if(status)

System.out.println("List contains the element 'E' ");

else

System.out.println("List doesn't contain the element 'E'");

// Number of elements in the linked list

int size = object.size();

System.out.println("Size of linked list = " + size);

// Get and set elements from linked list

Object element = object.get(2);

System.out.println("Element returned by get() : " + element);

object.set(2, "Y");

System.out.println("Linked list after change : " + object);

}

}

***Output:***

***Linked list : [D, A, E, B, C, F, G]***

***Linked list after deletion: [A, E, F]***

***List contains the element 'E'***

***Size of linked list = 3***

***Element returned by get() : F***

***Linked list after change : [A, E, Y]***

**Methods for Java LinkedList:**

1. int size(): It returns the number of elements in this list.
2. void clear(): It removes all of the elements from the list.
3. Object clone(): It is used to make the copy of an existing linked list.
4. Object set(int index, Object element): It is used to replace an existing element in the list with a new element.
5. boolean contains(Object element): It returns true if the element is present in the list.
6. boolean add(Object element): It appends the element to the end of the list.
7. void add(int index, Object element): It inserts the element at the position ‘index’ in the list.
8. boolean addAll(Collection C): It appends a collection to a Linked List.
9. boolean addAll(int index, Collection C): It appends a collection to a linked list at a specified position.
10. void addFirst(Object element): It inserts the element at the beginning of the list.
11. void addLast(Object element): It appends the element at the end of the list.
12. Object get(int index): It returns the element at the position ‘index’ in the list. It throws ‘IndexOutOfBoundsException’ if the index is out of range of the list.
13. Object getFirst(): It returns the first element of the Linked List.
14. Object getLast(): It returns the last element of the Linked List.
15. int indexOf(Object element): If element is found, it returns the index of the first occurrence of the element. Else, it returns -1.
16. int lastIndexOf(Object element): If element is found, it returns the index of the last occurrence of the element. Else, it returns -1.
17. Object remove(): It is used to remove and return the element from the head of the list.
18. Object remove(int index): It removes the element at the position ‘index’ in this list. It throws ‘NoSuchElementException’ if the list is empty.
19. boolean remove(Object O): It is used to remove a particular element from the linked list and returns a boolean value.
20. Object removeLast(): It is used to remove and return the last element of the Linked List.

Set in Java

Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.

Basically, Set is implemented by **HashSet, LinkedHashSet or TreeSet** (sorted representation).

Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

// Java code for adding elements in Set

import java.util.\*;

public class Set\_example

{

public static void main(String[] args)

{

// Set deonstration using HashSet

Set<String> hash\_Set = new HashSet<String>();

hash\_Set.add("Smita");

hash\_Set.add("For");

hash\_Set.add("Smita");

hash\_Set.add("Example");

hash\_Set.add("Set");

System.out.print("Set output without the duplicates");

System.out.println(hash\_Set);

// Set deonstration using TreeSet

System.out.print("Sorted Set after passing into TreeSet");

Set<String> tree\_Set = new TreeSet<String>(hash\_Set);

System.out.println(tree\_Set);

}

}

(Please note that we have entered a duplicate entity but it is not displayed in the output. Also, we can directly sort the entries by passing the unordered Set in as the parameter of TreeSet).

**Output:**

Set output without the duplicates[Smita, Example, For, Set]

Sorted Set after passing into TreeSet[Example, For, Smita, Set]

**Note:** As we can see the duplicate entry “Smita” is ignored in the final output, Set interface doesn’t allow duplicate entries.

Now we will see some of the basic operations on the Set i.e. Union, Intersection and Difference.

Let’s take an example of two integer Sets:

* [1, 3, 2, 4, 8, 9, 0]
* [1, 3, 7, 5, 4, 0, 7, 5]

**Union**  
In this, we could simply add one Set with other. Since the Set will itself not allow any duplicate entries, we need not take care of the common values.

**Expected Output:**

Union : [0, 1, 2, 3, 4, 5, 7, 8, 9]

**Intersection**  
We just need to retain the common values from both Sets.

**Expected Output:**

Intersection : [0, 1, 3, 4]

**Difference**  
We just need to remove all the values of one Set from the other.  
 **Expected Output:**

Difference : [2, 8, 9]

|  |
| --- |
| // Java code for demonstrating union, intersection and difference  // on Set  import java.util.\*;  public class SetExample  {      public static void main(String args[])      {          Set<Integer> a = new HashSet<Integer>();          a.addAll(Arrays.asList(new Integer[] {1, 3, 2, 4, 8, 9, 0}));          Set<Integer> b = new HashSet<Integer>();          b.addAll(Arrays.asList(new Integer[] {1, 3, 7, 5, 4, 0, 7, 5}));            // To find union          Set<Integer> union = new HashSet<Integer>(a);          union.addAll(b);          System.out.print("Union of the two Set");          System.out.println(union);            // To find intersection          Set<Integer> intersection = new HashSet<Integer>(a);          intersection.retainAll(b);          System.out.print("Intersection of the two Set");          System.out.println(intersection);            // To find the symmetric difference          Set<Integer> difference = new HashSet<Integer>(a);          difference.removeAll(b);          System.out.print("Difference of the two Set");          System.out.println(difference);      }  } |

**Output:**

Union of the two Set[0, 1, 2, 3, 4, 5, 7, 8, 9]

Intersection of the two Set[0, 1, 3, 4]

Difference of the two Set[2, 8, 9]

# LinkedHashSet in Java with Examples

A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

**Syntax:**

LinkedHashSet<String> hs = new LinkedHashSet<String>();

* Contains unique elements only like HashSet. It extends HashSet class and implements Set interface.
* Maintains insertion order.

Given below are the list of constructors supported by the LinkedHashSet:

1. **HashSet():** This constructor is used to create a default HashSet.
2. **HashSet(Collection C):** Used in initializing the HashSet with the eleements of the collection C
3. **LinkedHashSet(int size):** Used to intialize the size of the LinkedHashSet with the integer mentioned in the parameter.
4. **LinkedHashSet(int capacity, float fillRatio):** Can be used to initialize both the capacity and the fill ratio, also called the load capacity of the LinkedHashSet with the arguments mentioned in the parameter. When the number of elements exceeds the capacity of the hash set is multiplied with the fill ratio thus expanding the capacity of the LinkedHashSet

Below program explains the basic add and traversal operation of LinkedHashSet:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| import java.util.LinkedHashSet;  public class DemoLinkedHashSet {      public static void main(String[] args)      {          LinkedHashSet<String> linkedset = new LinkedHashSet<String>();          // Adding element to LinkedHashSet          linkedset.add("A");          linkedset.add("B");          linkedset.add("C");          linkedset.add("D");          // This will not add new element as A already exists          linkedset.add("A");          linkedset.add("E");          System.out.println("Size of LinkedHashSet = " + linkedset.size());          System.out.println("Original LinkedHashSet:" + linkedset);          System.out.println("Removing D from LinkedHashSet: " + linkedset.remove("D"));          System.out.println("Trying to Remove Z which is not "+ "present: " + linkedset.remove("Z"));          System.out.println("Checking if A is present=" +  linkedset.contains("A"));          System.out.println("Updated LinkedHashSet: " + linkedset);      }  }  **Output:**  Size of LinkedHashSet=5  Original LinkedHashSet:[A, B, C, D, E]  Removing D from LinkedHashSet: true  Trying to Remove Z which is not present: false  Checking if A is present=true  Updated LinkedHashSet: [A, B, C, E] TreeSet in Java TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface. It can also be ordered by a Comparator provided at set creation time, depending on which constructor is used. The TreeSet implements a NavigableSet interface by inheriting AbstractSet class. Few important features of TreeSet are as follows:   1. TreeSet implements the SortedSet interface so duplicate values are not allowed. 2. Objects in a TreeSet are stored in a sorted and ascending order. 3. TreeSet does not preserve the insertion order of elements but elements are sorted by keys. 4. TreeSet does not allow to insert Heterogeneous objects. It will throw classCastException at Runtime if trying to add hetrogeneous objects. 5. TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access and retrieval time. 6. TreeSet is basically implementation of a self-balancing binary search tree like [Red-Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/). Therefore operations like add, remove and search take O(Log n) time. And operations like printing n elements in sorted order takes O(n) time.   **Constructors of TreeSet class:**   1. **TreeSet t = new TreeSet();** This will create empty TreeSet object in which elements will get stored in default natural sorting order. 2. **TreeSet t = new TreeSet(Comparator comp);** This constructor is used when external specification of sorting order of elements is needed. 3. **TreeSet t = new TreeSet(Collection col);** This constructor is used when any conversion is needed from any Collection object to TreeSet object.      1. **TreeSet t = new TreeSet(SortedSet s)**; This constructor is used to convert SortedSet object to TreeSet Object.   **Synchronized TreeSet:** The implementation in a TreeSet is not synchronized in a sense that if multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be “wrapped” using the Collections.synchronizedSortedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set:  TreeSet ts = new TreeSet();  Set syncSet = Collections.synchronziedSet(ts);  Below program illustrates the basic opearation of a TreeSet:   |  | | --- | | // Java program to demonstrate insertions in TreeSet  import java.util.\*;    class TreeSetDemo {      public static void main(String[] args)      {          TreeSet<String> ts1 = new TreeSet<String>();            // Elements are added using add() method          ts1.add("A");          ts1.add("B");          ts1.add("C");            // Duplicates will not get insert          ts1.add("C");            // Elements get stored in default natural          // Sorting Order(Ascending)          System.out.println(ts1);      }  } |   Copy CodeRun on IDE  **Output:**  [A, B, C]  Two things must be kept in mind while creating and adding elements into a TreeSet:   * Firstly, insertion of null into a TreeSet throws NullPointerException because while insertion of null, it gets compared to the existing elements and null cannot be compared to any value. * Secondly, if we are depending on default natural sorting order, compulsory the object should be **homogeneous**and **comparable** otherwise we will get **RuntimeException:**ClassCastException  |  | | --- | | // Java code to illustrate StringBuffer  // class does not implements  // Comparable interface.  import java.util.\*;  public class TreeSetDemo {      public static void main(String[] args)   {          TreeSet<StringBuffer> ts = new TreeSet<StringBuffer>();          // Elements are added using add() method          ts.add(new StringBuffer("A"));          ts.add(new StringBuffer("Z"));          ts.add(new StringBuffer("L"));          ts.add(new StringBuffer("B"));          ts.add(new StringBuffer("O"));          // We will get RunTimeException :ClassCastException          // As StringBuffer does not implements Comparable interface          System.out.println(ts);      }  } |  * https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Screenshot-421-1.png   . **NOTE:**   1. An object is said to be comparable if and only if the corresponding class implements **Comparable interface**. 2. **String** class and all **Wrapper** classes already implements Comparable interface but StringBuffer class doesn’t implements Comparable interface.Hence we got ClassCastException in the above example. 3. For an empty tree-set, when trying to insert null as first value, one will get NPE from JDK 7.From 1.7 onwards null is not at all accepted by TreeSet. However upto JDK 6, null will be accepted as first value, but any if insertion of any more values in the TreeSet, will also throw NullPointerException. Hence it was considered as bug and thus removed in JDK 7.   **Methods of TreeSet class:**  TreeSet implements SortedSet so it has availability of all methods in Collection, Set and SortedSet interfaces. Following are the methods in Treeset interface.  1. void add(Object o): This method will add specified element according to some sorting order in TreeSet. Duplicate entires will not get added.  2. boolean addAll(Collection c): This method will add all elements of specified Collection to the set. Elements in Collection should be homogeneous otherwise ClassCastException will be thrown. Duplicate Entries of Collection will not be added to TreeSet.  3. void clear(): This method will remove all the elements.  4. boolean contains(Object o): This method will return true if given element is present in TreeSet else it will return false.  5. Object first(): This method will return first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.  6. Object last(): This method will return last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.  7. SortedSet headSet(Object toElement): This method will return elements of TreeSet which are less than the specified element.  8. SortedSet tailSet(Object fromElement): This method will return elements of TreeSet which are greater than or equal to the specified element.  9. SortedSet subSet(Object fromElement, Object toElement): This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive.  10. boolean isEmpty(): This method is used to return true if this set contains no elements or is empty and false for the opposite case.  11. Object clone(): The method is used to return a shallow copy of the set, which is just a simple copied set.  12. int size(): This method is used to return the size of the set or the number of elements present in the set.  13. boolean remove(Object o): This method is used to return a specific element from the set.  14. Iterator iterator(): Returns an iterator for iterating over the elements of the set.  15. Comparator comparator(): This method will return Comparator used to sort elements in TreeSet or it will return null if default natural sorting order is used. Map Interface in Java The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types. mapinterface Few characteristics of the Map Interface are:   1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the HashMap and LinkedHashMap, but some do not like the TreeMap. 2. The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not. 3. There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, TreeMap and LinkedHashMap.   https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Selection_030.png  **Why and When to use Maps?** 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 examples are:   * 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).   **Methods in Map Interface:**   1. public Object put(Object key, Object value): This method is used to insert an entry in this map. 2. public void putAll(Map map): This method is used to insert the specified map in this map. 3. public Object remove(Object key): This method is used to delete an entry for the specified key. 4. public Object get(Object key):This method is used to return the value for the specified key. 5. public boolean containsKey(Object key): This method is used to search the specified key from this map. 6. public Set keySet(): This method is used to return the Set view containing all the keys. 7. public Set entrySet(): This method is used to return the Set view containing all the keys and values.  |  | | --- | | // Java program to demonstrate working of Map interface  import java.util.\*;  public class HashMapDemo  {     public static void main(String args[])     {         Map< String,Integer> hm =                          new HashMap< String,Integer>();         hm.put("a", new Integer(100));         hm.put("b", new Integer(200));         hm.put("c", new Integer(300));         hm.put("d", new Integer(400));           // Returns Set view         Set< Map.Entry< String,Integer> > st = hm.entrySet();           for (Map.Entry< String,Integer> me:st)         {             System.out.print(me.getKey()+":");             System.out.println(me.getValue());         }     }  } |   Copy CodeRun on IDE  **Output:**  a:100  b:200  c:300  d:400 HashMap in Java **HashMap** is a part of Java’s collection since Java 1.2. It provides the basic implementation of Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. HashMap is known as HashMap because it uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents same String. A shorter value helps in indexing and faster searches. HashSet also uses HashMap internally. It internally uses a link list to store key-value pairs already explained in HashSet in detail and further articles. Few important features of HashMap are:   * HashMap is a part of java.util package. * HashMap extends an abstract class AbstractMap which also provides an incomplete implementation of Map interface. * It also implements Cloneable and Serializable interface. K and V in the above definition represent Key and Value respectively. * HashMap doesn’t allow duplicate keys but allows duplicate values. That means A single key can’t contain more than 1 value but more than 1 key can contain a single value. * HashMap allows null key also but only once and multiple null values. * This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. It is roughly similar to HashTable but is unsynchronized.   **Internal Structure of HashMap**  Internally HashMap contains an array of Node and a node is represented as a class which contains 4 fields :   1. int hash 2. K key 3. V value 4. Node next   It can be seen that node is containing a reference of its own object. So it’s a linked list. HashMap: [array](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/array.png)  Node : [node_hash_map](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/node_hash_map.png)    **Performance of HashMap**  Performance of HashMap depends on 2 parameters:   1. Initial Capacity 2. Load Factor   As already said, Capacity is simply the number of buckets whereas the Initial Capacity is the capacity of HashMap instance when it is created. The Load Factor is a measure that when rehashing should be done. Rehashing is a process of increasing the capacity. In HashMap capacity is multiplied by 2. Load Factor is also a measure that what fraction of the HashMap is allowed to fill before rehashing. When the number of entries in HashMap increases the product of current capacity and load factor the capacity is increased that is rehashing is done. If the initial capacity is kept higher then rehashing will never be done. But by keeping it higher it increases the time complexity of iteration. So it should be choosed very cleverly to increase the performance. The expected number of values should be taken into account to set initial capacity. Most generally preffered load factor value is 0.75 which provides a good deal between time and space costs. Load factor’s value varies between 0 and 1.  **Synchronized HashMap**  As it is told that HashMap is unsynchronized i.e. multiple threads can access it simultaneously. If multiple threads access this class simultaneously and at least one thread manipulates it structurally then it is necessary to make it synchronized externally. It is done by synchronizing some object which enzapsulates the map. If No such object exists then it can be wrapped around Collections.synchronizedMap() to make HashMap synchronized and avoid accidental unsynchronized access. As in following example:  Map m = Collections.synchronizedMap(new HashMap(...));  Now the Map m is synchronized.  Iterators of this class are fail-fast if any structure modification is done after creation of iterator, in any way except through the iterator’s remove method. In faliure of iterator it will throw ConcurrentModificationException.  **Constructors in HashMap**  HashMap provides 4 constructors and access modifier of each is public:   1. **HashMap() :** It is the default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75. 2. **HashMap(int initial capacity) :** It creates a HashMap instance with specified initial capacity and load factor 0.75. 3. **HashMap(int initial capacity, float loadFactor) :** It creates a HashMap instance with specified initial capacity and specified load factor. 4. **HashMap(Map map) :** It creates instance of HashMapwith same mappings as specified map.   **Example:**   |  | | --- | | // Java program to illustrate  // Java.util.HashMap  import java.util.HashMap;  import java.util.Map;    public class TestHashMap  {      public static void main(String[] args)      {            HashMap<String, Integer> map = new HashMap<>();            print(map);          map.put("vishal", 10);          map.put("sachin", 30);          map.put("vaibhav", 20);            System.out.println("Size of map is:- " + map.size());            print(map);          if (map.containsKey("vishal"))          {              Integer a = map.get("vishal");              System.out.println("value for key \"vishal\" is:- " + a);          }            map.clear();          print(map);      }        public static void print(Map<String, Integer> map)      {          if (map.isEmpty())          {              System.out.println("map is empty");          }            else          {              System.out.println(map);          }      }  } |   Copy CodeRun on IDE  **Output:**  map is empty  Size of map is:- 3  {vaibhav=20, vishal=10, sachin=30}  value for key "vishal" is:- 10  map is empty  **Time complexity of HashMap**  HashMap provides constant time complexity for basic operations, get and put, if hash function is properly written and it disperses the elements properly among the buckets. Iteration over HashMap depends on the capacity of HashMap and number of key-value pairs. Basically it is directly proportional to the capacity + size. Capacity is the number of buckets in HashMap. So it is not a good idea to keep high number of buckets in HashMap initially.  **Methods in HashMap**  1. void clear(): Used to remove all mappings from a map.  2. boolean containsKey(Object key): Used to return True if for a specified key, mapping is present in the map.  3. boolean containsValue(Object value): Used to return true if one or more key is mapped to a specified value.  4. Object clone(): It is used to return a shallow copy of the mentioned hash map.  5. boolean isEmpty(): Used to check whether the map is empty or not. Returns true if the map is empty.  6. Set entrySet(): It is used to return a set view of the hash map.  7. Object get(Object key): It is used to retrieve or fetch the value mapped by a particular key.  8. Set keySet(): It is used to return a set view of the keys.  9. int size(): It is used to return the size of a map.  10. Object put(Object key, Object value): It is used to insert a particular mapping of key-value pair into a map.  11. putAll(Map M): It is used to copy all of the elements from one map into another.  12. Object remove(Object key): It is used to remove the values for any particular key in the Map.  13. Collection values(): It is used to return a Collection view of the values in the HashMap. HashMap and TreeMap in Java HashMap and TreeMap are part of collection framework.  **HashMap**  java.util.HashMap class is a Hashing based implementation. In HashMap, we have a key and a value pair<Key, Value>.  HashMap<K, V> hmap = new HashMap<K, V>();  Let us consider below example where we have to count occurrences of each integer in given array of integers.  Input: arr[] = {10, 3, 5, 10, 3, 5, 10};  Output: Frequency of 10 is 3  Frequency of 3 is 2  Frequency of 5 is 2   |  | | --- | | /\* Java program to print frequencies of all elements using     HashMap \*/  import java.util.\*;    class Main  {      // This function prints frequencies of all elements      static void printFreq(int arr[])      {          // Creates an empty HashMap          HashMap<Integer, Integer> hmap =                       new HashMap<Integer, Integer>();            // Traverse through the given array          for (int i = 0; i < arr.length; i++)          {              Integer c = hmap.get(arr[i]);                // If this is first occurrence of element              if (hmap.get(arr[i]) == null)                 hmap.put(arr[i], 1);                // If elements already exists in hash map              else                hmap.put(arr[i], ++c);          }            // Print result          for (Map.Entry m:hmap.entrySet())            System.out.println("Frequency of " + m.getKey() +                               " is " + m.getValue());      }        // Driver method to test above method      public static void main (String[] args)      {          int arr[] = {10, 34, 5, 10, 3, 5, 10};          printFreq(arr);      }  } |   Copy CodeRun on IDE  Output:  Frequency of 34 is 1  Frequency of 3 is 1  Frequency of 5 is 2  Frequency of 10 is 3  **Key Points**   * HashMap does not maintain any order neither based on key nor on basis of value, If we want the keys to be maintained in a sorted order, we need to use TreeMap. * **Complexity**: get/put/containsKey() operations are O(1) in average case but we can’t guarantee that since it all depends on how much time does it take to compute the hash.   **Application:** HashMap is basically an implementation of hashing. So wherever we need hashing with key value pairs, we can use HashMap. For example, in Web Applications username is stored as a key and user data is stored as a value in the HashMap, for faster retrieval of user data corresponding to a username.    **TreeMap**  TreeMap can be a bit handy when we only need to store unique elements in a sorted order. Java.util.TreeMap uses a red-black tree in the background which makes sure that there are no duplicates; additionally it also maintains the elements in a sorted order.  TreeMap<K, V> hmap = new TreeMap<K, V>();  Below is TreeMap based implementation of same problem. This solution has more time complexity O(nLogn) compared to previous one which has O(n). The advantage of this method is, we get elements in sorted order.     |  | | --- | | /\* Java program to print frequencies of all elements using     TreeMap \*/  import java.util.\*;    public class TestTreeMap  {      // This function prints frequencies of all elements      static void printFreq(int arr[])      {          // Creates an empty TreeMap          TreeMap<Integer, Integer> tmap =                       new TreeMap<Integer, Integer>();            // Traverse through the given array          for (int i = 0; i < arr.length; i++)          {              Integer c = tmap.get(arr[i]);                // If this is first occurrence of element              if (tmap.get(arr[i]) == null)                 tmap.put(arr[i], 1);                // If elements already exists in hash map              else                tmap.put(arr[i], ++c);          }            // Print result          for (Map.Entry m:tmap.entrySet())            System.out.println("Frequency of " + m.getKey() +                               " is " + m.getValue());      }        // Driver method to test above method      public static void main (String[] args)      {          int arr[] = {10, 34, 5, 10, 3, 5, 10};          printFreq(arr);      }  } |   Copy CodeRun on IDE  Output:  Frequency of 3 is 1  Frequency of 5 is 2  Frequency of 10 is 3  Frequency of 34 is 1  **Key Points**   * For operations like add, remove, containsKey, time complexity is O(log n where n is number of elements present in TreeMap. * TreeMap always keeps the elements in a sorted(increasing) order, while the elements in a HashMap have no order. TreeMap also provides some cool methods for first, last, floor and ceiling of keys.   **Overview:**   1. HashMap implements Map interface while TreeMap implements SortedMap interface. A Sorted Map interface is a child of Map. 2. HashMap implements Hashing, while TreeMap implements Red-Black Tree(a Self Balancing Binary Search Tree). Therefore all differences between Hashing and Balanced Binary Search Tree apply here. 3. Both HashMap and TreeMap have their counterparts HashSet and TreeSet. HashSet and TreeSet implement Set interface. In HashSet and TreeSet, we have only key, no value, these are mainly used to see presence/absence in a set. For above problem, we can’t use HashSet (or TreeSet) as we can’t store counts. An example problem where we would prefer HashSet (or TreeSet) over HashMap (or TreeMap) is to print all distinct elements in an array.   collection_interfaces  **R** Differences between TreeMap, HashMap and LinkedHashMap in Java **TreeMap, HashMap and LinkedHashMap: What’s Similar?**   * All offer **a key->value** map and a way to iterate through the keys. The most important distinction between these classes is the time guarantees and the ordering of the keys. * All three classes HashMap, TreeMap and LinkedHashMap implements **java.util.Map** interface, and represents mapping from unique key to values.   **Key Points**   1. **HashMap:** HashMap offers **0(1)** lookup and insertion. If you iterate through the keys, though, the ordering of the keys is essentially arbitrary. It is implemented by an array of linked lists. **Syntax:** 2. **public class HashMap extends AbstractMap**   **implements Map,Cloneable, Serializable**   * + A HashMap contains values based on the key.   + It contains only unique elements.   + It may have one null key and multiple null values.   + It maintains **no order**.  1. **LinkedHashMap:**LinkedHashMap offers **0(1)** lookup and insertion. Keys are ordered by their insertion order. It is implemented by doubly-linked buckets. **Syntax:**     **public class LinkedHashMap extends HashMap**  **0implements Map**   * + A LinkedHashMap contains values based on the key.   + It contains only unique elements.   + It may have one null key and multiple null values.   + It is same as HashMap instead **maintains insertion order**.  1. **TreeMap:** TreeMap offers **O(log N)** lookup and insertion. Keys are ordered, so if you need to iterate through the keys in sorted order, you can. This means that keys must implement the Comparable interface. TreeMap is implemented by a Red-Black Tree. **Syntax:** 2. **public class TreeMap extends AbstractMap implements**   **NavigableMap, Cloneable, Serializable**   * + A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.   + It contains only unique elements.   + It cannot have null key but can have multiple null values.   + It is same as HashMap instead **maintains ascending order(Sorted using the natural order of its key**).  1. **Hashtable:**“Hashtable” is the generic name for hash-based maps. **Syntax:** 2. **public class Hashtable extends Dictionary implements**   **Map, Cloneable, Serializable**   * + A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.   + It contains only unique elements.   + It may have not have any null key or value.   + It is synchronized.   + It is a legacy class.  |  | | --- | | // Java program to print ordering  // of all elements using HashMap  import java.util.\*;  import java.lang.\*;  import java.io.\*;  public class DemoTreeMap  {      // This function prints ordering of all elements      static void insertAndPrint(AbstractMap<Integer, String> map)      {          int[] array= {1, -1, 0, 2,-2};          for (int x: array)          {              map.put(x, Integer.toString(x));          }          for (int k: map.keySet())          {              System.out.print(k + ", ");          }      }        // Driver method to test above method      public static void main (String[] args)      {          HashMap<Integer, String> map = new HashMap<Integer, String>();          insertAndPrint(map);      }  } |   Copy CodeRun on IDE  Output of HashMap:  -1, 0, 1, -2, 2,  // ordering of the keys is essentially arbitrary (any ordering)  Output of LinkedHashMap:  1, -1, 0, 2, -2,  // Keys are ordered by their insertion order  Output of TreeMap:  -2, -1, 0, 1, 2,  // Keys are in sorted order  **Comparison Table**https://cdncontribute.geeksforgeeks.org/wp-content/uploads/comparisonTable.png  **Real Life Applications**   1. Suppose you were creating a mapping of names to Person objects. You might want to periodically output the people in alphabetical order by name. A TreeMap lets you do this. 2. A TreeMap also offers a way to, given a name, output the next 10 people. This could be useful for a “More”function in many applications. 3. A LinkedHashMap is useful whenever you need the ordering of keys to match the ordering of insertion. This might be useful in a caching situation, when you want to delete the oldest item. 4. Generally, unless there is a reason not to, you would use HashMap. That is, if you need to get the keys back in insertion order, then use LinkedHashMap. If you need to get the keys back in their true/natural order, then use TreeMap. Otherwise, HashMap is probably best. It is typically faster and requires less overhead.   Differences between HashMap and HashTable in Java  HashMap and Hashtable store key/value pairs in a hash table. When using a Hashtable or HashMap, we specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.  Sample Java code.   |  | | --- | | // A sample Java program to demonstrate HashMap and HashTable  import java.util.\*;  import java.lang.\*;  import java.io.\*;    /\* Name of the class has to be "Main" only if the class is public. \*/  class Ideone  {      public static void main(String args[])      {          //----------hashtable -------------------------          Hashtable<Integer,String> ht=new Hashtable<Integer,String>();          ht.put(101," ajay");          ht.put(101,"Vijay");          ht.put(102,"Ravi");          ht.put(103,"Rahul");          System.out.println("-------------Hash table--------------");          for (Map.Entry m:ht.entrySet()) {              System.out.println(m.getKey()+" "+m.getValue());          }            //----------------hashmap--------------------------------          HashMap<Integer,String> hm=new HashMap<Integer,String>();          hm.put(100,"Amit");          hm.put(104,"Amit");  // hash map allows duplicate values          hm.put(101,"Vijay");          hm.put(102,"Rahul");          System.out.println("-----------Hash map-----------");          for (Map.Entry m:hm.entrySet()) {              System.out.println(m.getKey()+" "+m.getValue());          }      }  } |   Copy CodeRun on IDE  Output:    -------------Hash table--------------  103 Rahul  102 Ravi  101 Vijay  -----------Hash map-----------  100 Amit  101 Vijay  102 Rahul  104 Amit  **Hashmap vs Hashtable** 1. HashMap is non synchronized. It is not-thread safe and can’t be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads. 2. HashMap allows one null key and multiple null values whereas Hashtable doesn’t allow any null key or value. 3. HashMap is generally preferred over HashTable if thread synchronization is not needed  Why HashTable doesn’t allow null and HashMap does? To successfully store and retrieve objects from a HashTable, the objects used as keys must implement the hashCode method and the equals method. Since null is not an object, it can’t implement these methods. HashMap is an advanced version and improvement on the Hashtable. HashMap was created later.  Following is the difference between LinkedHashMap and LinkedHashSet: https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Capture-70.jpg  **Important :** Keeping the insertion order in both LinkedHashmap and LinkedHashset have additional associated costs, both in terms of spending additional CPU cycles and needing more memory. If you do not need the insertion order maintained, it is recommended to use the lighter-weight HashSet and HashMapinstead. |