Java Collection Framework, Collections

Collection is a group of individual objects

It is an interface which contains child Interface and their classes.

Whenever we want to group individual objects we have arrays in java, there are several problems with arrays

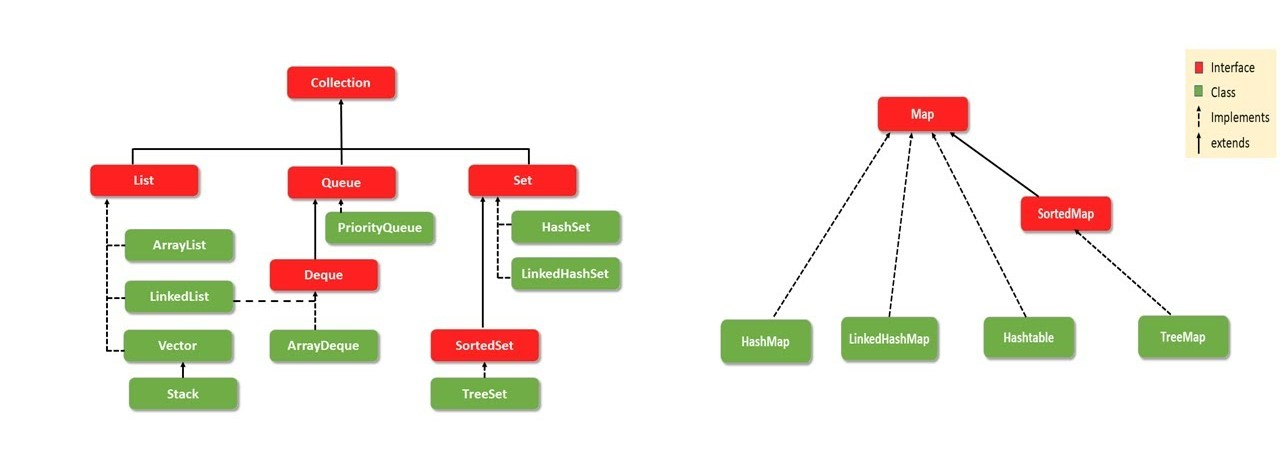
1. Fixed in size
2. Sored only homogeneous objects
3. They don’t used underlying data structure i.e. they don’t have predefined methods(e.g if we want to sort array then we need to write separate logic for that we cant use .sort method)

To overcome with problems in arrays we have Collection.

|  |  |
| --- | --- |
| Array | Collection |
| Fixed in size | Not fixed in size |
| Stored homogenous data | Stored heterogeneous data |
| No used of underlying DS | Use of underlying DS |

Collection in java is a framework that provides architecture to store and manipulate group of objects.

java.util package contains all the interface and classes for the collection framework.



Methods of collection(I)

1. add(object)
2. addAll(collection)
3. remove(object)
4. removeAll(collection)
5. retailAll(collection) --- remove all except specified.
6. size()
7. clear()
8. contains(object)
9. containsAll(collection)
10. isEmpty()
11. equals(object)
12. hashcode() ---- return hashcode of collection

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods

1. hashnext() ---returns true if the iterator has more elements
2. next() --returns the element and moves the cursor pointer to the next element.
3. remove() ---removes the last elements returned by the iterator.

List Interface

Child I of collection I.

Stored ordered collection of objects.

Can have duplicate values.

Also stored Null value.

Index based methods method, useful for searching than insertion/ deletion operations.

Implemented by ArrayList, LinkedList, Vector and Stack.

Declaration

1. **public** **interface** List<E> **extends** Collection<E>

Methods

Other than collection(I)

1. get(index)
2. indexOf(object)
3. set(index, element)
4. sort() – Collections.sort() -- present inside collections utility class.
5. Shuffle() –Collection.shuffle() --

Implementation

1. List <data-type> list1= **new** ArrayList()/LinkedList()/Stack()/Vector;
2. Eg:: List<String> list=**new** ArrayList<String>();
3. List<Book> list=**new** ArrayList<Book>();

ListIterator Interface

used to traverse the element in a backward and forward direction.

Declaration

1. **public** **interface** ListIterator<E> **extends** Iterator<E>

Methods

Other than collection I

hasPrevious() ---returns true if this list iterator has more elements while traversing the list in the reverse direction.

Previous() --returns the previous element in the list and moves the cursor position backward.

previousIndex() --returns the index of the element that would be returned

ArrayList class.

**ArrayList** class uses a dynamic [*array*](https://www.javatpoint.com/array-in-java) for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array.

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java).
* Java ArrayList allows random access because the array works on an index basis.
* In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
* We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example.

ArrayList<**int**> al = ArrayList<**int**>(); // does not work

ArrayList<Integer> al = **new** ArrayList<Integer>(); // works fine

Declaration

**public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

Method other than present inside collection and list I

1. ensureCapacity(int requiredCapacity) --enhance the capacity of an ArrayList instance.
2. clone() --used to return a shallow copy of an ArrayList.
3. ReplaceAll(new elements) --replace all the elements from the list with the specified.
4. trimToSize() --trim the capacity of this ArrayList instance to be the list's

list current size.

Implementation

ArrayList list=**new** ArrayList();//creating old non-generic arraylist

ArrayList<String> list=**new** ArrayList<String>();//creating new generic arraylist

Iterating elements of ArrayList

1) Iterator interface.

2) for-each loop.

3) ListIterator interface.

4) for loop.

5) forEach() method.

6) forEachRemaining() method.

Size and Capacity of ArrayList

Default size is 0 , default capacity is 10

Therefore when 11th element is added into Arraylist new object will create with capacity 11 and previous ArrayList will copy into new ArrayList object and previous object will eligible for garbage collection(GC).

LinkedList class.

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure.

Used to implements List and Deque interfaces.

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

Declaration

**public** **class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

Methods

Other than method present inside list, collection and deque I

1) addFirst(element)

2) allLast(element)

3) getFirst()

4) getLas()

5) peek() --retrieves the first element of a list

6) peekFirst() --retrieves the first element of a list or returns null if a list is empty

7) peekLast() --retrieves the last element of a list or returns null if a list is empty.

8) removeFirstOccurrence(Object)

9) removeLastOccurrence(Object)

Stack and dequeue methods

1)pop() --pops an element from the stack represented by a list.

2)push() --pushes an element onto the stack represented by a list.

Implementation

LinkedList list=**new** LinkedList();//creating old non-generic arraylist

LinkedList<String> list=**new** LinkedList<String>();//creating new generic arraylist

List<Book> list=**new** LinkedList<Book>();

ArrayList VS LinkedList

ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

|  |  |
| --- | --- |
| ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |
| 5) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |
| 6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized. |
| 7) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list interface. |

* When the rate of addition or removal rate is more than the read scenarios, then go for the LinkedList. On the other hand, when the frequency of the read scenarios is more than the addition or removal rate, then ArrayList takes precedence over LinkedList.
* Since the elements of an ArrayList are stored more compact as compared to a LinkedList; therefore, the ArrayList is more cache-friendly as compared to the LinkedList. Thus, chances for the cache miss are less in an ArrayList as compared to a LinkedList. Generally, it is considered that a LinkedList is poor in cache-locality.
* Memory overhead in the LinkedList is more as compared to the ArrayList. It is because, in a LinkedList, we have two extra links (next and previous) as it is required to store the address of the previous and the next nodes, and these links consume extra space. Such links are not present in an ArrayList.

Set Interface

1) It is the collection of unordered set of elements which doesn't allow us to store the duplicate items.

2) Store at most one null value.

3) implemented by HashSet, LinkedHashSet, and TreeSet.

Implementation

Set<data-type> s1 = **new** HashSet<data-type>();

Set<data-type> s2 = **new** LinkedHashSet<data-type>();

Set<data-type> s3 = **new** TreeSet<data-type>();

HashSet()

Java HashSet class is used to create a collection that uses a hash table for storage. It means it uses Hash Table as

Underlying DS.

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.
* The initial default capacity of HashSet is 16, and the load factor is 0.75.

Declaration

**public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

Constructors

1) HashSet()

2) HashSet(int capacity)

3) HashSet(int capacity, float loadFactor)

Methods

1) add(object)

2) clear()

3) contains(object)

4) isEmpty()

5) remove(object)

6) size()

HashSet doesn’t have any predefined methods.

Whatever methods available inside set(I), collection(I), collections(c) are available for HashSet.

Implementation

1. HashSet<String> hs=**new** HashSet<>();

HashSet hs=new HashSet(100, float(0.90));

LinkedHashSet()

Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface.

inherits the HashSet class and implements the Set interface.

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operations and permits null elements.
* Java LinkedHashSet class is non-synchronized.
* Java LinkedHashSet class maintains insertion order.

Declaration

**public** **class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Serializable

LinkedHashSet<String> set=**new** LinkedHashSet();

Constructors

1) LinkedHashSet(int capacity) -- used to initialize the capacity of the linked hash set to the given

value.

2) LinkedHashSet(int capacity, float fillRatio) -- initialize both the capacity and the fill ratio.

All the methods which are available in HashSet are also available in LinkedHshSet.

TreeSet

Queue Interface

used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

Prior to processing elements.

Insertion order preserved and duplicates are allowed.

Null objects are not allowed in queue.

LinkedList and PriorityQueue classes are used to implement Queue I. (Not thread safe)

PriorityBlockingQueue class is Thread safe.

* As discussed earlier, FIFO concept is used for insertion and deletion of elements from a queue.
* The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.
* PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.
* The NullPointerException is raised, if any null operation is done on the BlockingQueues.
* Those Queues that are present in the *util*package are known as Unbounded Queues.
* Those Queues that are present in the *util.concurrent*package are known as bounded Queues.
* All Queues barring the Deques facilitates removal and insertion at the head and tail of the queue; respectively. In fact, deques support element insertion and removal at both ends.

Cant add null elements in queue.

Declaration

1. **public** **interface** Queue<E> **extends** Collection<E>

Methods

Other than present in collection I.

1) add(element) -- used to add element return true if element is inserted, raised exception if it is not

succesful.

2) offer(element) -- used to add element return true if element is inserted , return false is not inseted.

3) remove() --used to retrieves and removes the head element of this queue. If queue is empty then raised

exception.

4) poll() -- used to retrieves and removes the head element of this queue, if empty then return false.--

5) element() --retrieves, the head element of this queue, raised exception is not retrieve.

6) peek() --retrieves, the head element of this queue, return false if not retrieve.

PriorityQueue

It is the class used to implement queue interface, process the objects on the basis of priority.

Insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

Declaration

1. **public** **class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

Implementation

1. PriorityQueue<String> queue=**new** PriorityQueue<String>();
2. LinkedList<String> queue=new LinkedList<String>();
4. Thread safe
5. PriorityBlokingQueue()

The**PriorityBlockingQueue** is an unbounded blocking queue that uses the same ordering rules

as class [**PriorityQueue**](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/) and supplies blocking retrieval operations. Since it is unbounded, adding elements

may sometimes fail due to resource exhaustion resulting in **[OutOfMemoryError](https://www.geeksforgeeks.org/understanding-outofmemoryerror-exception-java/)**.

Declaration

public class PriorityBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable

Implementation

PriorityBlockingQueue<E> pbq = new PriorityBlockingQueue<E>();

Deque Interface

Known as doubly ended queue

supports the addition as well as the removal of elements from both ends of the data structure.

Dequeue can be used as Stack.

stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue. As

deque supports both, either of the mentioned operations can be performed on it.

Therefore deque can be use for both Stack and Queue.

Declaration

**public** **interface** Deque<E> **extends** Queue<E>

Implementation

Implemented by LinkedList and ArrayDequeue

Deque<String> deque = **new** ArrayDeque<String>();

Deque<String> deque =new LinkedList<String>();

Methods

1)add(element) -- This method is used to add an element at the tail of the queue. If the Deque is capacity is ended

And no space is left for insertion, it returns an IllegalStateException. The function returns true on

Successful insertion.

2)addFirst(element) -- add element at the head, and if capacity is ended then, IllegalStateException and return on

Success.

3)addLast(element) -- add element at the tail, and if capacity is ended then, IllegalStateException and return on

Success.

4)contains(element)

5)element() -- retrieve head of the deque.

6)getFirst()

7)getLast()

8)offer() --used to add element at the tail, return null if capacity over.

9)offerFirst() -- used to add element at the head, return null if capacity over.

10)offerLast() -- used to add element at the tail, return null if capacity over.

11)peek() --used to retrieve element at the head, return null if deque empty.

12)peekFirst() -- used to retrieve element at the head, return null if deque empty.

13)peekLast() -- used to retrieve element at the tail, return null if deque empty.

14)poll() --used to retrieve and remove element at the head, return null if deque empty.

15)pollFist() -- used to retrieve and remove element at the head, return null if deque empty.

16)pollLast() -- used to retrieve and remove element at the tail, return null if deque empty.

17)pop() --used to remove element at the head.

18)push() -- used to add element at the head.

19)removeFirst() – used to remove element from the head of the queue.

20)removeLast() – used to remove element from the tail of the queue.

21)size()



ArrayDeque

It is the linear collection that supports element insertion and removal at both ends.

Used to implement deque I and Queue I.

provides a way to apply resizable-array in addition to the implementation of the Deque interface. It is also known as ***Array Double Ended Queue*** or ***Array Deck***. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.

Initial capacity is 16.

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

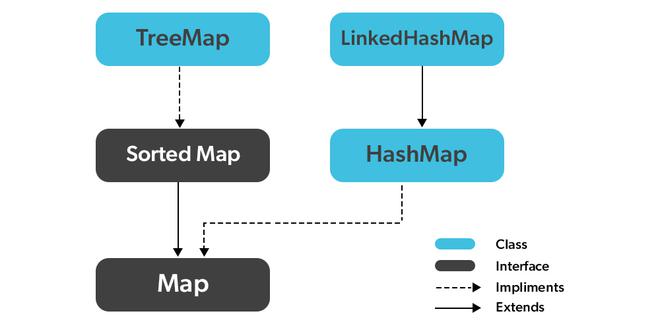
Declaration

**public** **class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Serializable

Implementation

ArrayDeque<E> dq = new ArrayDeque<E>();

Map Interface.



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.

Key cant be duplicate

Values can be duplicates

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

Key and value collectively call as entry.

Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

Use of map I

* 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).

Implementation

|  |  |
| --- | --- |
| [HashMap](https://www.javatpoint.com/java-hashmap) | HashMap is the implementation of Map, but it doesn't maintain any order. |
| [LinkedHashMap](https://www.javatpoint.com/java-linkedhashmap) | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| [TreeMap](https://www.javatpoint.com/java-treemap) | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

Methods

1) put(key, value) --used to insert an entry in the map.

2) putAll(map) --used to insert the specified map in the map.

3) putIfAbsent(key, value) --inserts the specified value with the specified key in the map only if it is not already

Specified.

4) remove(Object key) --used to delete an entry for the specified key.

5) remove(key, value) --removes the specified values with the associated specified keys from the map.

6) clear()

7) containsValue(value)

8) containsKey(key)

9) equals(value) --checks the equality between two maps.

10) isEmpty()

11) replace(K key, V value) -- replace value for the given key.

12) replace(key, oldValue, newValue)

13) size()

14) containsValue(Object value)

15) containsKey(Object key)

16) keyset()

17) values()

18) get(key)

19) hashcode() -- return hashcode value.

Map.Entry Interface.

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name.

It return key value pair.

Methods of Entry I.

1)getKey()

2)getValue()

3)setValue()

4)entrySet() --return all entries as a set.

HashMap class

Java **HashMap** class implements the Map interface,

Allowed to store in key and value pair objects.

Easy to perform operation likes updation and insertion.

Doesn’t maintain insertion order.

* Java HashMap contains values based on the key.
* Java HashMap contains only unique keys.
* Java HashMap may have one null key and multiple null values.
* Java HashMap is non synchronized.
* Java HashMap maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.
* Insertion order is not maintain.

Declaration

1. **public** **class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

Constructors

|  |  |
| --- | --- |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map<? extends K,? extends V> m) | It is used to initialize the hash map by using the elements of the given Map object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value, capacity. |
| HashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and load factor of the hash map by using its arguments. |

Implementation

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

Methods

1)clear --Removes all of the mappings from this map.

2)clone() --return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

3)containsKey(key) --Returns true if this map contains a mapping for the specified key.

4)containsValue(value) --Returns true if this map maps one or more keys to the specified value.

5)entrySet() --Returns a Set view of the mappings contained in this map.

6)get(key) --Returns the value to which the specified key.

7)isEmpty()

8)keyset() --return the set of all keys

9)remove(key) -- remove value of the specify key.

10)size()

11)values() --Returns a Collection view of the values contained in this map.

12)put(key, value) –insert new entry into map if key already contain then it replace the value for particular key.

13)putAll()

Methods inherits from java.util.AbstractMap class

1)equals() -- it check equality between tow map.//map1.equals.map2

2)hashcode() --return hashcode value for map(it is the integer value associated with each object in java)

3)toString() --return string representation of map.

Methods inherits from java.util.map()

1)equals()

2)forEach()

3)hashCode()

4)putIfAbsent(key, value)

5)remove(key, value)

6)replace(key, value)

7)replace(key, oldValue, newValue)

Internal Structure of HashMap

HashMap use the hashing technique

It is the process of converting object into integer value, therefore it helps in indexing and faster searching.

HashMap contains array of nodes, it uses array and LinkedList Data Structure internally for storing key and value pairs.

Internally HashMap contains an array of Node and a node is represented as a class that contains 4 fields:

1. int hash
2. K key
3. V value
4. Node next

LinkedHashMap

The **LinkedHashMap** **Class** is just like [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) 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.

LinkedHashMap maintained the track of insertion order.

* A LinkedHashMap contains values based on the key. It implements the Map interface and extends the HashMap class.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is non-synchronized.
* It is the same as HashMap with an additional feature that it maintains insertion order. For example, when we run the code with a HashMap, we get a different order of elements.

Declaration

public class LinkedHashMap<K,​V> extends HashMap<K,​V> implements Map<K,​V>

Constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedHashMap() | It is used to construct a default LinkedHashMap. |
| LinkedHashMap(int capacity) | It is used to initialize a LinkedHashMap with the given capacity. |
| LinkedHashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and the load factor. |
| LinkedHashMap(int capacity, float loadFactor, boolean accessOrder) | It is used to initialize both the capacity and the load factor with specified ordering mode. |
| LinkedHashMap(Map<? extends K,? extends V> m) | It is used to initialize the LinkedHashMap with the elements from the given Map class m. |

Implementation

Methods

1)containsValue(value)

2)get(key)

3)keyset()

4)values()

Synchronized LinkedHashMap

HashTable

HashTable class implements Map interface,

* A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* Java Hashtable class contains unique elements.
* Java Hashtable class doesn't allow null key or value.
* Java Hashtable class is synchronized.
* The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Declaration

**public** **class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

Implementation

Hashtable<Integer, String> ht1 = new Hashtable<>();

Constructors

|  |  |
| --- | --- |
| Hashtable() | It creates an empty hashtable having the initial default capacity and load factor. |
| Hashtable(int capacity) | It accepts an integer parameter and creates a hash table that contains a specified initial capacity. |
| Hashtable(int capacity, float loadFactor) | It is used to create a hash table having the specified initial capacity and loadFactor. |
| Hashtable(Map<? extends K,? extends V> t) | It creates a new hash table with the same mappings as the given Map. |

Methods

1)clear()

2)clone() --create shallow copy of the hashtable // hashtable.clone()

3)contains(value)

4)containsKey(key)

5)containsValue(value)

6)elements() -- return enumeration of the values present in the hashtable. //

7)equals()

8)isEmpty()

9)hashCode()

10)put(key, value)

11)remove(key)

12)size()

13)toString() --convert map into string separated by ,

14)get(key)

15)keys() --return enumeration of the keys

16)values() --return collection view of values.

Difference between HashTable and HashMap

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

Comparable Interface

Java comparable interface used to sort/ order the objects of the user-defined class.

This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

Method

**compareTo(Object obj):**

It is used to compare the current object with the specified object.

* positive integer, if the current object is greater than the specified object.
* negative integer, if the current object is less than the specified object.
* zero, if the current object is equal to the specified object.

1. String objects
2. Wrapper class objects
3. User-defined class objects

Comparable interface is used to sort user-defined class based on their objects by comparing.

Eg:

Comparator Interface

**Java Comparator interface** is used to order the objects of a user-defined class.

Present inside java.util class

Contains two methods

compare(Object obj1,Object obj2)

equals(Object element).

It help to sort user-defined interface similar to collection.sort() method but override the .sort() method for user-defined class and help to sort them

|  |  |
| --- | --- |
| **Method** | **Description** |
| public int compare(Object obj1, Object obj2) | It compares the first object with the second object. |
| public boolean equals(Object obj) | It is used to compare the current object with the specified object. |