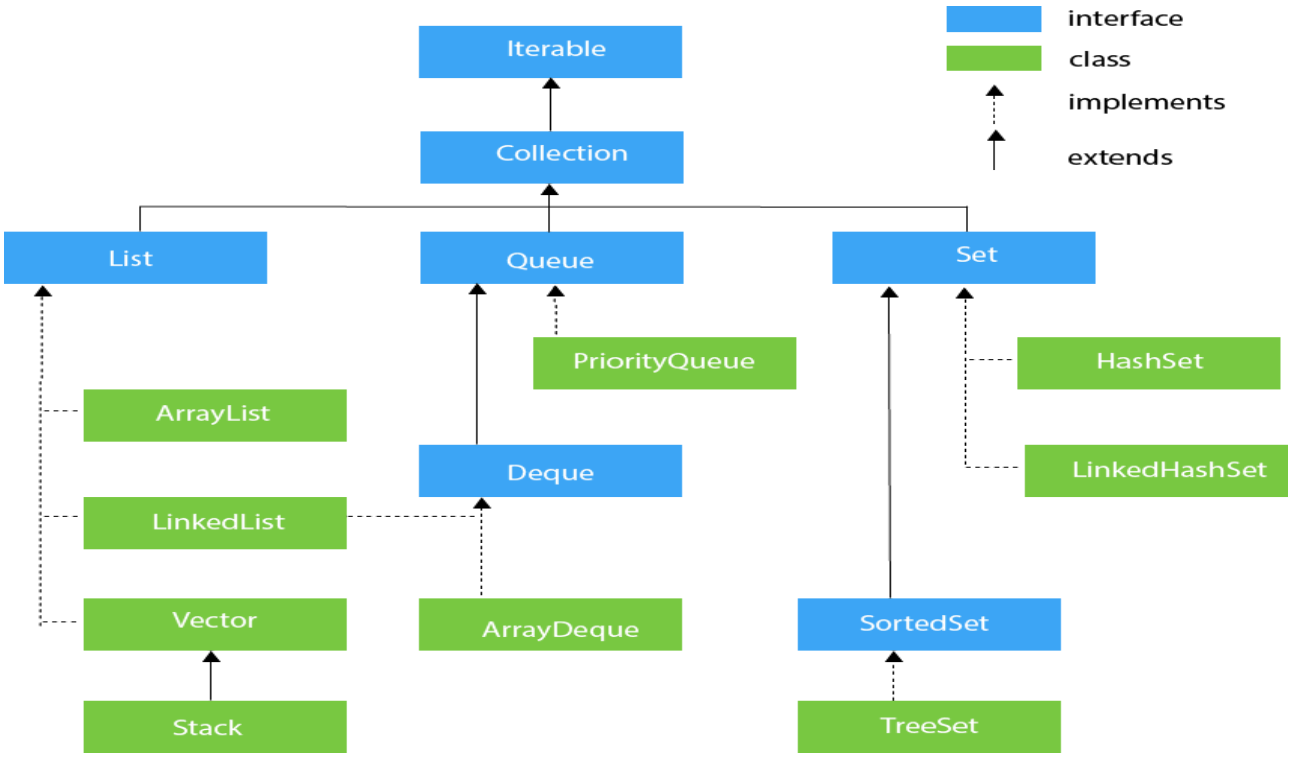
Iterable
Collection Interface



Collection is an Interface of collection framework.

Collections(java.util.) is a **class** having a predefined utlis method(sort...etc) which we will use in the collection interface.
Ex.
ArrayList a = new ArrayList();
Collections.sort(a);

Type of Collection interface(List,Set, Queues)

1)List

It is the child interface of the collection interface.
The insertion order is preserved.
Insertion order preserved i.e., They appear in the same order in which we inserted.
Duplicate elements are allowed.

Classes which implement List Interface.
i**) ArrayList ii) LinkedList iii) Vector**

2**)Set**
It is also the child interface of the collection interface.
The insertion order is not preserved.
Duplicate values are not allowed.

Classes which implement set Interface.
i)HashSet ii)LinkedHashSet

3) Queue
It is also the child interface of collection Interface.
Prior to process(priority wise will execute)

Classes which implements Queue Interface
PriorityQueue

Map Interface

It is not a child interface of collection interface.
Here the data/object is in the form of a key-value pair.
The **keys** are unique and not duplicate.
The **values** can be duplicate.

Classes which implement Map Interface
i)HashMap ii)LinkedHashMap

Few Basic Common methods of Collection interface(List,Set, Queue)

i)add(object/element e)- It will add a specific object to the collection.
ii)addAll(Collection c) - It will add a group of objects to the collection.
iii)remove (object o)- It will remove a specific object from the collection.
iv) removAll(Collection c) - It will remove a group of objects from the collection.
v) retainAll(Collection c)- Except this group of objects and it will remove the rest of the objects.
Vi).Clear (It will clear all objects from the collection)
Vii)isEmpty(It will check whether the collection is empty or not)
Viii)Size(It will give the size of collection)

ix) contains(object o) It will check whether an object is present in collection or not.
X)ContainsAll(it will check whether the group of objects present in the collection or not).
Xi)toArray(collection c)- it will convert collection to array.

List

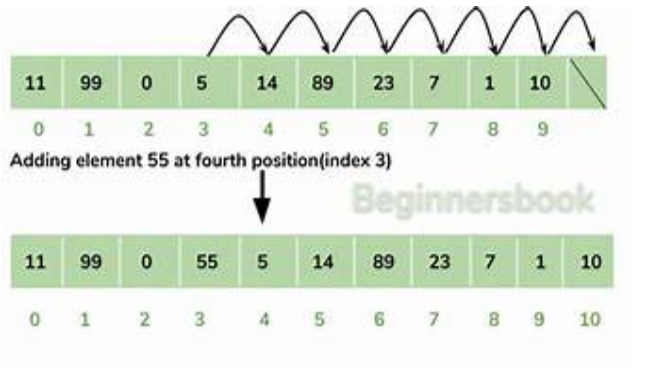
Apart from Common methods , List has its own methods.

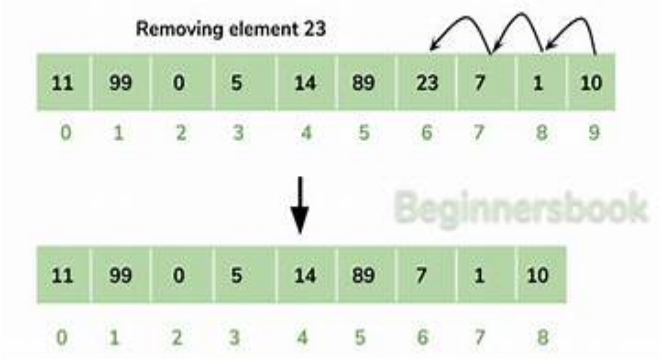
i) add(index, object) - will add object anywhere in the collection using index
Ii)addAll(index, object) -It will add group of objects anywhere in the collection using index
iii remove (index) - it will remove a specific object.
iv)get(index)- Will get a specified object based on index value.
V)set(index,new object) -It will replace the new object with the existing object.

These methods are like abstract methods and method definitions are available in the collection interface and List . But these are implemented in ArrayList and LinkedList.
It **accepts null** values and we can add as many null values as you like..

ArrayList

The default size initially created for ArrayList is 10
We can use ArrayList when we want to **add duplicate values and order should be preserved.**
It **accepts null** values and we can add as many null values as you like..
When you want to perform **frequent retrieve operations** then prefer ArrayList as we can fetch the data easily using index values.
We **shouldn't prefer** ArrayList when you want to perform more insertion/deletion operations. As in arraylist if we add a value or delete a new value the remaining value has to be shifted to either previous or next index values vice-versa.
EX ;





- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.

1)

```
package com.arraylistdemo;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
//
public class ArrayListDemo1 {
	public static void main(String[] args) {
		// 1. Declartion of arraylist
		// ArrayList a = new ArrayList();// Heterogeneous data we can store here
		// ArrayList<String> string = new ArrayList<String>();//Homogeneous data we
can
		// store
```

```
// <Integer> integer= new ArrayList<Integer>();
// List b= new ArrayList(); // ArrayList is a Java Class implemented Using the
// List interface
ArrayList al = new ArrayList();
// 2. adding elements to the ArrayList
al.add(100);
al.add("raju");
al.add(14.60);
al.add('B');
al.add(true);
al.add(null);// It accepts null values and we can add as many null values as you
like..

System.out.println(al); // insertion order is preserved
// 3.Size(It will give number of elements in arraylist)
int alSize = al.size();
System.out.println("number of elements in arraylist:" + alSize);
// 4.remove
al.remove(1); // removing element using index number
System.out.println(" After removing element using index number:" + al);
al.remove(true);// removing using object/ element directly
System.out.println(" After removing element using element directly:" + al);
// 5.insert a new element using index along with object
// add(index,object)
al.add(2, "ramesh");
System.out.println(" after insertion:" + al);
// 6.Retrieve a specific value/elememt using index number
System.out.println("getting specific value:" + al.get(1));
// 7.replace/ change existing element in arraylist
// al.set(index number, new value)
al.set(1, 15.66);
System.out.println("after replacing element:" + al);
// 8.searching , will use contains.Returns true/false
System.out.println(al.contains("ramesh")); // true
System.out.println(al.contains(100)); // true
System.out.println(al.contains("king")); // false
// 9.isEmpty
// It will check whether arraylist is empty or not
// If arraylist is empty it will return true otherwise return false
System.out.println(al.isEmpty()); // false
// 10.we can read the data / print the data in below methods
```

```
		// i) using for loop
		System.out.println("Reading the elements using for loop");
		for (int i = 0; i < al.size(); i++) {
			System.out.println(al.get(i));
		}
		// ii) using for each loop
		System.out.println("Reading the elements using for each loop");
		for (Object b : al) {
			System.out.println(b);
		}
		// iii) using iterator() method available in Iterator interface.
		// Iterator interface is parent of collection interface
		System.out.println("Reading the elements using iterator method");
		Iterator it = al.iterator();
		while (it.hasNext()) { // it will return true if the iterator has elements otherwise it
will return
                                                        // false
			System.out.println(it.next());// it.next(): it will print that element and
immediately it will go to the
                                                                    // next element
		}
	}
}
```

Output

```
[100, raju, 14.6, B, true, null]
number of elements in arraylist:6
After removing element using index number:[100, 14.6, B, true, null]
After removing element using element directly:[100, 14.6, B, null]
after insertion:[100, 14.6, ramesh, B, null]
getting specific value:14.6
after replacing element:[100, 15.66, ramesh, B, null]
true
true
false
false
Reading the elements using for loop
100
15.66
ramesh
```

```
B
null
Reading the elements using for each loop
100
15.66
ramesh
B
null
Reading the elements using iterator method
100
15.66
ramesh
B
null
```

2)

```
package com.arraylistdemo;
import java.util.ArrayList;
import java.util.Collections;
public class ArrayListDemo2 {
	public static void main(String[] args) {
		// arraylist 1
		ArrayList al = new ArrayList();
		al.add("A");
		al.add("C");
		al.add("B");
		al.add("F");
		al.add("E");
		al.add("D");
		System.out.println(al);
		// arraylist2
		ArrayList al2 = new ArrayList();
		al2.add(100);
		al2.add(200);
		System.out.println(al2);
		// adding one arraylist to another arraylist uding addAll()
		// addAll(collection)
		al2.addAll(al);
		System.out.println("after adding arraylist to another arraylist:" + al2);
		// removeAll method
```

```
                al2.removeAll(al);
                System.out.println("removing arraylist from another arraylist:" + al2);
                // sorting the arraylist elements
                // Collections class has few commons methods and we can use them in any
                // collection interface like list,set and queue .Collections class also
                // available in java.util only
                // Collections.sort()
                System.out.println("Elements in the arraylist al:" + al);
                Collections.sort(al);
                System.out.println("Elements in the arraylist after sorting al:" + al);
                // sorting in reverse order
                Collections.sort(al, Collections.reverseOrder());
                System.out.println("Elements in the arraylist in reverseorder al:" + al);
                // Shuffle- Collections.shuffle()
                Collections.shuffle(al);
                System.out.println("Elements in the arraylist after shuffle order al:" + al);
        }
}
```

Output

```
[A, C, B, F, E, D]
[100, 200]
after adding arraylist to another arraylist:[100, 200, A, C, B, F, E, D]
removing arraylist from another arraylist:[100, 200]
Elements in the arraylist al:[A, C, B, F, E, D]
Elements in the arraylist after sorting al:[A, B, C, D, E, F]
Elements in the arraylist in reverseorder al:[F, E, D, C, B, A]
Elements in the arraylist after shuffle order al:[E, B, C, A, D, F]
```

3)

```
package com.arraylistdemo;
import java.util.ArrayList;
import java.util.Arrays;
public class ArrayListDemo3 {
        public static void main(String[] args) {
                String arr[] = { "Rahul", "Ravi", "Roja" };
                for (String s : arr) {
                        System.out.println(s);
                }
```

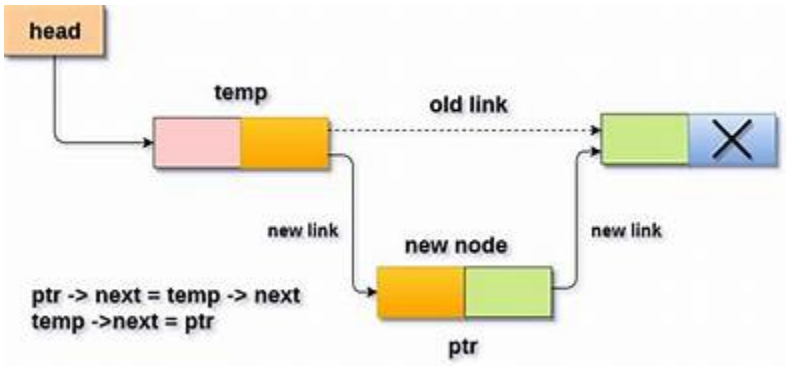
```
            // converting array to arraylist
            ArrayList al = new ArrayList(Arrays.asList(arr));
            System.out.println(al);
        }
}
```

Output

Rahul
Ravi
Roja
[Rahul, Ravi, Roja]

LinkedList

It is a class which implements **List Interface**.
It also implements **DeQueue Interface**.
It **accepts null** values and we can add as many null values as you like..
and there is no default size for linked list.
When we want to **add duplicate values and order should be preserved.** When you want to perform **frequent insertion and deletion operations** on a list then prefer LinkedList. Here we have nodes with addresses so linking nodes and removing nodes is easy and there is no need to shift data as like in arrayList.



We **shouldn't prefer** LinkedList when we have more retrieval.

Methods
Note: We can use whatever the methods available in List and Collections interface.,
add(x)
addAll(Collection c)
remove(obj)
removeAll(Collection c)
get(obj)
set(x,obj)....etc

Specific methods for linked list

LinkedList is used to implement Stack(FILO) and Queue(FIFO).

addFirst(Obj)
addLast(obj)
removeFirst()
removeLast()
getFirst()
getLast()

```
package com.linkedlistdemo;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
public class LinkedListDemo1 {
	public static void main(String[] args) {
		// Declare Linked List
		// LinkedList l = new LinkedList(); // it stores heterogeneous data
		// LinkedList<Integer> l = new LinkedList<Integer>(); // it stores homogeneous
		// data
		// LinkedList<String> l = new LinkedList<String>();
		// List l = new LinkedList();
		LinkedList l = new LinkedList();
		// add elements to linkedlist (it will add end of linkedlist by default)
		l.add(100);
		l.add("hi");
		l.add(true);
		l.add(null);// // It accepts null values and we can add as many null values as you
like.
		l.add(12.45);
		System.out.println(l);
		// size
		System.out.println("Number of elements in linked list=" + l.size());
		// remove
		l.remove(4);// index values
		System.out.println("After removing element, new list =" + l);
		// insert/ adding elements at a specified position
		l.add(2, "55.99");
		System.out.println("After inserting element, new list =" + l);
		// Retrieving value using index
		System.out.println("retrieve: " + l.get(1));
```

```
                // replace value
                l.set(4, "rani");
                System.out.println("After replacing element, new list =" + l);
                // contains(), if value present in linked list then it will return true otherwise
                // false
                System.out.println(l.contains("ramesh")); // false
                System.out.println(l.contains("55.99")); // true
                // isEmpty
                System.out.println(l.isEmpty());
                // reading the data 3 ways
                // i) using for loop
                System.out.println("Reading the elements using for loop");
                for (int i = 0; i < l.size(); i++) {
                        System.out.println(l.get(i));
                }
                // ii) using for each loop
                System.out.println("Reading the elements using for each loop");
                for (Object b : l) {
                        System.out.println(b);
                }
                // iii) using iterator() method available in Iterator interface.
                // Iterator interface is parent of collection interface
                System.out.println("Reading the elements using iterator method");
                Iterator it = l.iterator();
                while (it.hasNext()) { // it will return true if the iterator has elements otherwise it
will return
                                                                // false
                        System.out.println(it.next());// it.next(): it will print that element and
immediately it will go to the
                                                                        // next element
                }
        }
}
```

Output

```
[100, hi, true, null, 12.45]
Number of elements in linked list=5
After removing element, new list =[100, hi, true, null]
After inserting element, new list =[100, hi, 55.99, true, null]
retrieve: hi
```

```
After replacing element, new list =[100, hi, 55.99, true, rani]
false
true
false
Reading the elements using for loop
100
hi
55.99
true
rani
Reading the elements using for each loop
100
hi
55.99
true
rani
Reading the elements using iterator method
100
hi
55.99
true
rani
```

2)

```
package com.linkedlistdemo;
import java.util.Collections;
import java.util.LinkedList;
public class LinkedListDemo2 {
	public static void main(String[] args) {
		LinkedList l = new LinkedList();
		l.add("X");
		l.add("Y");
		l.add("Z");
		l.add("A");
		l.add("B");
		l.add("C");
		System.out.println("linkedlist:" + l);
		LinkedList new_l = new LinkedList();
		// addAll
		new_l.addAll(l);
```

```
                System.out.println("addAll" + new_l);
                // removeAll
                new_l.removeAll(l);
                System.out.println("removeAll" + new_l);
                // sorting
                System.out.println("Before sorting linked list:" + l);
                // Collections.sort(collection)
                Collections.sort(l);
                System.out.println("After sorting linked list:" + l);
                // sorting in reverse order
                Collections.sort(l, Collections.reverseOrder());
                System.out.println("Elements in the arraylist in reverseorder al:" + l);
                // Shuffle- Collections.shuffle()
                Collections.shuffle(l);
                System.out.println("Elements in the arraylist after shuffle order al:" + l);
        }
}
```

Output

```
linkedlist:[X, Y, Z, A, B, C]
addAll[X, Y, Z, A, B, C]
removeAll[]
Before sorting linked list:[X, Y, Z, A, B, C]
After sorting linked list:[A, B, C, X, Y, Z]
Elements in the arraylist in reverseorder al:[Z, Y, X, C, B, A]
Elements in the arraylist after shuffle order al:[X, A, Y, C, B, Z]
```

3)

```
package com.linkedlistdemo;
import java.util.LinkedList;
public class LinkedListDemo3 {
        public static void main(String[] args) {
                LinkedList l = new LinkedList<>();
                l.add("dog");
                l.add("cat");
                l.add("dog");
                l.add("horse");
                System.out.println(l);
                l.addFirst("zeebra");
```

```
                l.addLast("Tiger");
                System.out.println(l);
                System.out.println(l.getFirst());
                System.out.println(l.getLast());
                System.out.println(l.removeFirst());
                System.out.println(l.removeLast());
                System.out.println(l);
        }
}
```

Output

```
[dog, cat, dog, horse]
[zeebra, dog, cat, dog, horse, Tiger]
zeebra
Tiger
zeebra
Tiger
[dog, cat, dog, horse]
```

3) Vector

The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the ArrayList class.

Java Vector vs. ArrayList

In Java, both ArrayList and Vector implements the List interface and provide the same functionalities. However, there exist some differences between them.
The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called **ConcurrentModificationException** is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.

However, in array lists, methods are not synchronized. Instead, it uses the Collections.synchronizedList() method that synchronizes the list as a whole.

Note: It is recommended to use ArrayList in place of Vector because vectors are less efficient.

Creating a Vector
Here is how we can create vectors in Java.

```
Vector<Type> vector = new Vector<>();
```

Here, Type indicates the type of a linked list. For example,

```
// create Integer type linked list
Vector<Integer> vector= new Vector<>();

// create String type linked list
Vector<String> vector= new Vector<>();
```

Methods

add(element) - adds an element to vectors
add(index, element) - adds an element to the specified position
addAll(vector) - adds all elements of a vector to another vector
get(index) - returns an element specified by the index
iterator() - returns an iterator object to sequentially access vector elements
remove(index) - removes an element from specified position
removeAll() - removes all the elements
clear() - removes all elements. It is more efficient than removeAll()
set()-changes an element of the vector
size()-returns the size of the vector
toArray()-converts the vector into an array
toString()-converts the vector into a String
contains()-searches the vector for specified element and returns a boolean result

```
package com.vectordemo;
import java.util.Iterator;
import java.util.Vector;
public class VectorDemo {
	public static void main(String[] args) {
		Vector<String> v = new Vector<>();
		// Using the add() method
		v.add("Dog");
		v.add("Horse");
		v.add("Cat");
		// Using index number
		v.add(2, "Cat");
		System.out.println("Vector: " + v);
```

```
                // Using addAll()
                Vector<String> animals = new Vector<>();
                animals.add("Crocodile");
                animals.add("Cat");
                animals.add("cow");
                animals.addAll(v);
                System.out.println("New Vector: " + animals);
                // Using get()
                String element = animals.get(2);
                System.out.println("Element at index 2: " + element);
                // Using iterator()
                System.out.print("using iterator method");
                Iterator<String> iterate = animals.iterator();
                while (iterate.hasNext()) {
                        System.out.println(iterate.next());
                }
                // Using remove()
                String element1 = animals.remove(1);
                System.out.println("Removed Element: " + element1);
                System.out.println("New Vector: " + animals);
                // Using clear()
                animals.clear();
                System.out.println("Vector after clear(): " + animals);
        }
}
```

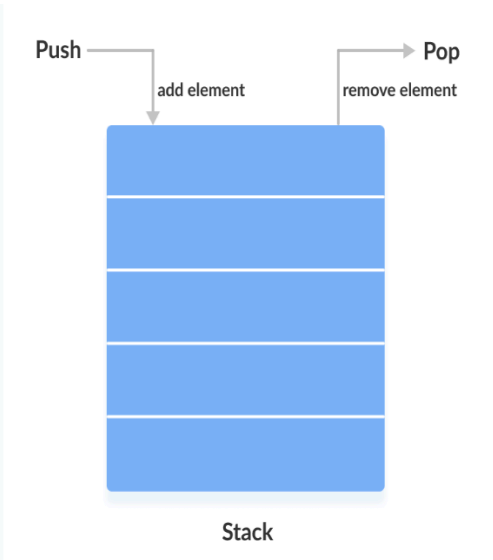
Output

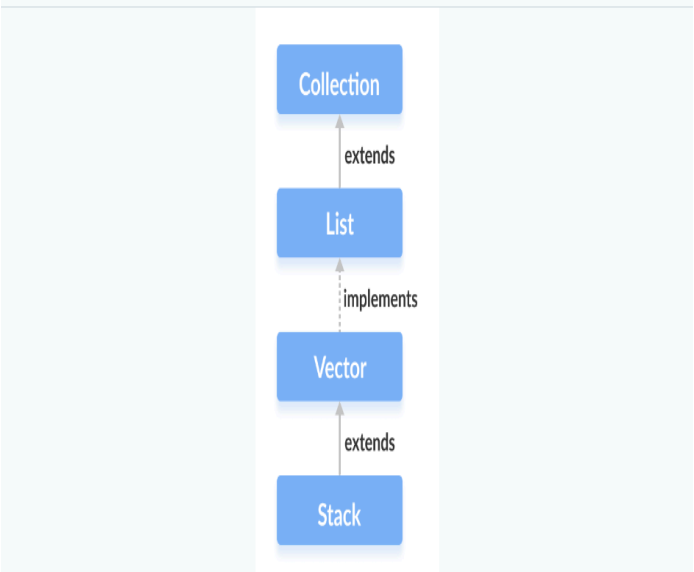
```
Element at index 2: cow
using iterator methodCrocodile
Cat
cow
Dog
Horse
Cat
Cat
Removed Element: Cat
New Vector: [Crocodile, cow, Dog, Horse, Cat, Cat]
Vector after clear(): []
```

Stack

The Java collections framework has a class named Stack that provides the functionality of the stack data structure.

The Stack class extends the Vector class.
In stack, elements are stored and accessed in Last In First Out manner. That is, elements are added to the top of the stack and removed from the top of the stack.





Stack<Type> stacks = new Stack<>();
Stack<Integer> stacks1= new Stack<>();
Since Stack extends the Vector class, it inherits all the methods of Vector. To learn about different Vector methods.
The Stack class includes 5 more methods that distinguish it from Vector.
push() Method
pop() Method
peek() Method
search() Method
empty() Method
Note: The Stack class provides the direct implementation of the stack data structure. However, it is recommended not to use it. Instead, use the ArrayDeque class (implements the Deque interface) to implement the stack data structure in Java.

```
package com.vectordemo;
import java.util.Stack;
```

```java
//LIFO last in fast
public class StackDemo {
	public static void main(String[] args) {
		Stack<String> animals = new Stack<>();
		// push() Add elements to Stack
		// LIFO last in fast
		animals.push("Dog");
		animals.push("Horse");
		animals.push("Cat");
		animals.push("cow");
		animals.push("goat");
		animals.push("Cat");
		System.out.println("Stack: " + animals);
		// pop()
		// Remove element stacks
		String element = animals.pop();
		System.out.println("Removed Element: " + element);//// LIFO last in fast
		// peek() method
		// Access element from the top
		String element1 = animals.peek();
		System.out.println("Element at top: " + element1);
		// search()
		// Search an element
		int position = animals.search("Horse");
		System.out.println("Position of Horse: " + position);
		// empty
		// Check if stack is empty
		boolean result = animals.empty();
		System.out.println("Is the stack empty? " + result);
	}
}
```

Output

```
Stack: [Dog, Horse, Cat, cow, goat, Cat]
Removed Element: Cat
Element at top: goat
Position of Horse: 4
Is the stack empty? false
```

Set

The Set Interface is present in java.util package and extends the Collection interface. It is an unordered collection of objects in which duplicate values cannot be stored.

- HashSet allows only one null value

We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

i) HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only and doesn't allow duplicate values .
- HashSet allows only one 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.

By default,

- the capacity of the hash set will be 16
- the load factor/Fill Ratio will be 0.75

```
// HashSet with default capacity and load factor

HashSet<Integer> numbers1 = new HashSet<>();
```

Once the 75% of the hashset is stored with elements/objects then a new hashset will be created automatically .

Load Capacity = Initial Capacity * Load Factor

= 16 * 0.75

= 12

We can initialize the size of the hashset and load factor of the hashset as mentioned below.

```
/ HashSet with 8 capacity and 0.6 load factor

HashSet<Integer> numbers = new HashSet<>(8, 0.6);
```

Here, we have created a hash set named numbers.

Notice, the part new HashSet<>(8, 0.6). Here, the first parameter is capacity, and the second parameter is loadFactor.

Hash set doesn't have any own methods but we can use whatever the methods are available in the set and collection interface .

Declaration of hashset.

i) Default Constructor

```
HashSet<Type> set = new HashSet<>();
```

ii) With Initial Capacity

```
HashSet<Type> set = new HashSet<>(initialCapacity);
```

- initialCapacity is the initial capacity of the HashSet.

iii) With Initial Capacity and Load Factor

```
HashSet<Type> set = new HashSet<>(initialCapacity, loadFactor);
```

iv) With Another Collection

```
Collection<Type> collection = ...;

HashSet<Type> set = new HashSet<>(collection);
```

or

i) `Set<Type> s1 = new HashSet<>();`

ii) `Set<Type> s1 = new HashSet<>(initialCapacity);`

iii) `Set<Type> s1 = new HashSet<>(initialCapacity, loadFactor);`

iv)

```
Collection<Type> collection = ...;

Set<Type> s1 = new HashSet<>(collection);

List<String> list = Arrays.asList("Apple", "Banana", "Cherry");

Set<String> s1 = new HashSet<>(list);
```

In general, the first approach (`Set<Type> s1 = new HashSet<>();`) is preferred because it allows for greater flexibility and better adherence to coding principles. By coding to the interface (Set), your code becomes more maintainable and adaptable.

Use the second approach (`HashSet<Type> s1 = new HashSet<>();`) when you need to work with features or methods specific to HashSet that are not available in the Set interface.

Ex . methods

Here there are no index concepts as elements stored based on hashcode or randomly.

add(value)

addAll(Collection c)
remove(value)
removeAll(Collection c)
contains
containsAll
isEmpty
….etc

We don’t have any method to sort or shuffle a hashset directly as there are no index concepts/ elements that are not stored in sequence order.
We can sort the hashset externally by converting it to an arraylist or linkedlist.

Examples

1)

```
package com.hashsetdemo;
import java.util.HashSet;
import java.util.Iterator;
public class HashsetDemo1 {
	public static void main(String[] args) {
		/*
		 * 1. Declaration of Hashset HashSet hs = new HashSet(); // default capacity 16
		 * and load factor 0.75 HashSet hs1 = new HashSet(100); // initial capacity 100
		 * HashSet hs2= new HashSet(100,(float)0.90); // along with initial capacity ,
		 * load factor
		 *
		 * type of objects HashSet<Integer> hs = new HashSet<Integer>();
HashSet<String>
		 * hs = new HashSet<String>();
		 */
		HashSet hs = new HashSet();
		// adding objects or elements into HashSet
		hs.add(100);
		hs.add(-90);
```

```
hs.add(0);
hs.add('A');
hs.add("ram");
hs.add(12.67);
hs.add(true);
hs.add(null);//it accepts only one null value
//lhs.add(null);

hs.add(100);
System.out.println(hs); // insertion order is not preserved
// remove
hs.remove(-90);
System.out.println(hs); // after removing object from hashset.
// here we don't have method to get specific value from hashset as like get
// method in list.
// we don't have have method to modify existing data also as like set method in
// list.
// Size(It will give number of elements in hashset)
System.out.println(hs.size());
// contains() method . searching , will use contains.Returns true/false
System.out.println(hs.contains(100));// true
System.out.println(hs.contains(89)); // false
System.out.println(hs.contains(null)); // true
// isEmpty()
// It will check whether hashset is empty or not
// If hashset is empty it will return true otherwise return false
System.out.println(hs.isEmpty());
// we can read the data / print the data in below methods
// using for loop we can't read as there is not get method
// i) using for each loop
System.out.println("Reading the elements using for each loop");
for (Object o : hs) {
        System.out.println(o);
}
// 2.using iterator() method available in Iterator interface.
// Iterator interface is parent of collection interface
System.out.println("Reading the elements using iterator method");
Iterator itr = hs.iterator();
while (itr.hasNext()) { // it will return true if the iterator has elements otherwise it
will return
```

```
                                                        // false
                        System.out.println(itr.next());// it.next(): it will print that element and
immediately it will go to the
                        // next element
                }
        }
}
```

Output:

```
[0, null, A, 100, 12.67, -90, ram, true]
[0, null, A, 100, 12.67, ram, true]
7
true
false
true
false
Reading the elements using for each loop
0
null
A
100
12.67
ram
true
Reading the elements using iterator method
0
null
A
100
12.67
ram
true
```

2)

```
package com.hashsetdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
```

```
import java.util.List;
public class HashSetDemo2 {
	public static void main(String[] args) {
		HashSet<Integer> evenNumber = new HashSet<Integer>();
		evenNumber.add(2);
		evenNumber.add(4);
		evenNumber.add(6);
		evenNumber.add(8);
		System.out.println("HashSet Value:" + evenNumber);
		HashSet<Integer> number = new HashSet<Integer>();
		// addAll method
		number.addAll(evenNumber);
		number.add(10);
		number.add(14);
		System.out.println("New HashSet Value:" + number);
		// removeAll
		number.removeAll(evenNumber);
		System.out.println("after removeAll method:" + number);
		// remove
		number.remove(14);
		System.out.println("after remove method:" + number);
		/*
		 * We don't have any method to sort or shuffle a hashset directly as there are
		 * no index concepts/ elements that are not stored in sequence order. We can
		 * sort the hashset externally by converting it to an arraylist or linkedlist.
		 */
		// Creating a HashSet
		HashSet<String> set = new HashSet<String>();
		// Adding elements into HashSet using add()
		set.add("geeks");
		set.add("practice");
		set.add("contribute");
		set.add("ide");
		System.out.println("Original HashSet: " + set);
		// Sorting HashSet using List
		List<String> list = new ArrayList<String>(set);
		Collections.sort(list);
		// Print the sorted elements of the HashSet
		System.out.println("HashSet elements " + "in sorted(ascending/natural) order " +
"using List: " + list);
```

```
                Collections.sort(list, Collections.reverseOrder());
                System.out.println("HashSet elements reverseOrder (descending)oder: " + list);
        }
}
```

Output

```
HashSet Value:[2, 4, 6, 8]
New HashSet Value:[2, 4, 6, 8, 10, 14]
after removeAll method:[10, 14]
after remove method:[10]
Original HashSet: [practice, geeks, contribute, ide]
HashSet elements in sorted(ascending/natural) order using List: [contribute, geeks, ide, practice]
HashSet elements reverseOrder (descending)oder: [practice, ide, geeks, contribute]
```

3)

```
package com.hashsetdemo;
import java.util.HashSet;
public class HashSetDemo3 {
        public static void main(String[] args) {
                // Union : it will collect unique elements from the hashsets
                // Intersection (intersection is a type of set operation where the resultant set
                // will contain elements that are present in both of the sets.)
                // Difference
                HashSet<Integer> set1 = new HashSet<Integer>();
                set1.add(1);
                set1.add(2);
                set1.add(3);
                set1.add(4);
                set1.add(5);
                System.out.println("Hash Set1:" + set1);
                HashSet<Integer> set2 = new HashSet<Integer>();
                set2.add(3);
                set2.add(4);
                set2.add(5);
                System.out.println("Hash Set2:" + set2);
                // union: it will collect unique elements from the hashsets and give as a result
                //set1.addAll(set2);
```

```
                //System.out.println("Union:" + set1);
                // intersection (common elements):the resultant set will contain elements that
                // are present in
                // both of the sets
                 //set1.retainAll(set2);
                 //System.out.println("intersection:" + set1);
                // Difference : it will remove common elements in hashsets
                //set1.removeAll(set2);
                //System.out.println("Difference:" + set1);

    //subset: if set2 elements has present in set 1 then s2 is subset of s1

                //System.out.println("subset :"+ set1.containsAll(set2));

        }
}
```

Output

1)
Hash Set1:[1, 2, 3, 4, 5]
Hash Set2:[3, 4, 5]
Union:[1, 2, 3, 4, 5]

2)Hash Set1:[1, 2, 3, 4, 5]
Hash Set2:[3, 4, 5]
intersection:[3, 4, 5]

3)
Hash Set1:[1, 2, 3, 4, 5]
Hash Set2:[3, 4, 5]
Difference:[1, 2]
4)
Hash Set1:[1, 2, 3, 4, 5]
Hash Set2:[3, 4, 5]
subset :true

2) LinkedHashSet

- The Java LinkedHashSet class contains unique elements only like HashSet and doesn't allow duplicate values.
- Java LinkedHashSet class maintains insertion order.
- LinkedHashSet allows the inclusion of null elements, but like HashSet, it can only contain one null element.
- The only major difference between hashset and Linkedhashset is whereas LinkedHashSet class maintains insertion order . but HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.

  Both hashset andLlinkedhashset use the same methods.

  The LinkedHashSet class extends the HashSet class, which implements the Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

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

Declaration of LinkedHashSet :

1. LinkedHashSet(): This constructor is used to create a default HashSet

```
LinkedHashSet<E> hs = new LinkedHashSet<E>();
```

2. LinkedHashSet(Collection C): Used in initializing the HashSet with the elements of the collection C.

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(Collection c);
```

3. LinkedHashSet(int size): Used to initialize the size of the LinkedHashSet with the integer mentioned in the parameter.

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(int size);
```

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.

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(int capacity, int fillRatio);
```

or

```
Set<Type> s1 = new LinkedHashSet<>();

Set<Type> s1 = new LinkedHashSet<>(initialCapacity);

Set<Type> s1 = new LinkedHashSet<>(initialCapacity, loadFactor);

Collection<Type> collection = ...;

Set<Type> s1 = new LinkedHashSet<>(collection);
```

(List<String> list = Arrays.asList("Apple", "Banana", "Cherry"); Set<String> s1 = new LinkedHashSet<>(list);)

In general, the first approach (Set<Type> s1 = new LinkedHashSet<>();) is preferred because it allows for greater flexibility and better adherence to coding principles. By coding to the interface (Set), your code becomes more maintainable and adaptable.

Use the second approach (LinkedHashSet<Type> s1 = new LinkedHashSet<>();) when you need to work with features or methods specific to HashSet that are not available in the Set interface.

1)

```
package com.linkedhashsetdemo;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;
public class LinkedHashSetDemo1 {
```

```
public static void main(String[] args) {
        /*
         * 1. Declaration of LinkedHashSet lhs= new LinkedHashSet(); // default capacity
         * 16 and load factor 0.75 LinkedHashSet lhs1 = new LinkedHashSet(100); //
         * initial capacity 100 LinkedHashSet lhs2= new LinkedHashSet(100,(float)0.90);
         * // along with initial capacity , load factor
         *
         * type of objects LinkedHashSet<Integer> lhs = new LinkedHashSet<Integer>();
         * LinkedHashSet<String> lhs = new LinkedHashSet<String>();
         */
        LinkedHashSet lhs = new LinkedHashSet();
        // Set lhs1 = new LinkedHashSet();
        lhs.add(1);
        lhs.add('A');
        lhs.add("ram");
        lhs.add(12.30);
        lhs.add(null);
        System.out.println(lhs); // insertion order is preserved
        // remove
        System.out.println(lhs.remove(-90));// removing the element which is not present
        lhs.remove(1);
        System.out.println(lhs); // after removing object from LinkedHashset.
        // here we don't have method to get specific value from LinkedHashset as like
        // get
        // method in list.
        // we don't have have method to modify existing data also as like set method in
        // list.
        // Size(It will give number of elements in LinkedHashset)
        System.out.println(lhs.size());
        // contains() method . searching , will use contains.Returns true/false
```

```
		System.out.println(lhs.contains('A'));// true
		System.out.println(lhs.contains(99)); // false
		System.out.println(lhs.contains(null)); // true
		// isEmpty()
		// It will check whether LinkedHashset is empty or not
		// If LinkedHashset is empty it will return true otherwise return false
		System.out.println(lhs.isEmpty());
		// we can read the data / print the data in below methods
		// using for loop we can't read as there is not get method
		// i) using for each loop
		System.out.println("Reading the elements using for each loop");
		for (Object o : lhs) {
			System.out.println(o);
		}
		// 2.using iterator() method available in Iterator interface.
		// Iterator interface is parent of collection interface
		System.out.println("Reading the elements using iterator method");
		Iterator itr = lhs.iterator();
		while (itr.hasNext()) { // it will return true if the iterator has elements otherwise it
will return
								// false
			System.out.println(itr.next());// it.next(): it will print that element and
immediately it will go to the
			// next element
		}
	}
}
```

Output

```
[1, A, ram, 12.3, null]
false
```

[A, ram, 12.3, null]

4

true

false

true

false

Reading the elements using for each loop

A

ram

12.3

null

Reading the elements using iterator method

A

ram

12.3

null

2)

```
package com.linkedhashsetdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedHashSet;
import java.util.List;
public class LinkedHashSetDemo2 {
	public static void main(String[] args) {
		LinkedHashSet<Integer> evenNumber = new LinkedHashSet<Integer>();
		evenNumber.add(2);
		evenNumber.add(6);
		evenNumber.add(6);
```

```
        evenNumber.add(8);
        evenNumber.add(4);
        System.out.println("LinkedHashSet order is preserved:" + evenNumber); //order is preserved.
        LinkedHashSet<Integer> number = new LinkedHashSet<Integer>();
        // addAll method
        number.addAll(evenNumber);
        number.add(10);
        number.add(14);
        System.out.println("New LinkedHashSet Value:" + number);
        // removeAll
        number.removeAll(evenNumber);
        System.out.println("after removeAll method:" + number);
        // remove
        number.remove(14);
        System.out.println("after remove method:" + number);
        /*
         * We don't have any method to sort or shuffle a LinkedHashset directly as there are
         * no index concepts/ elements that are not stored in sequence order. We can
          * sort the LinkedHashset externally by converting it to an ArrayList or LinkedList.
         */
        // Creating a LinkedHashSet
        LinkedHashSet<String> set = new LinkedHashSet<String>();
        // Adding elements into LinkedHashSet using add()
        set.add("geeks");
        set.add("practice");
        set.add("contribute");
        set.add("ide");
```

```
                System.out.println("Original LinkedHashSet: " + set);
                // Sorting LinkedHashSet using List
                List<String> list = new ArrayList<String>(set);
                Collections.sort(list);
                // Print the sorted elements of the LinkedHashSet
                System.out.println("LinkedHashSet elements " + "in sorted(ascending/natural)
order " + "using List: " + list);
                Collections.sort(list, Collections.reverseOrder());
                System.out.println("LinkedHashSet elements reverseOrder (descending)oder: " +
list);
        }
}
```

Output

```
LinkedHashSet order is preserved:[2, 6, 8, 4]
New LinkedHashSet Value:[2, 6, 8, 4, 10, 14]
after removeAll method:[10, 14]
after remove method:[10]
Original LinkedHashSet: [geeks, practice, contribute, ide]
LinkedHashSet elements in sorted(ascending/natural) order using List: [contribute, geeks, ide,
practice]
LinkedHashSet elements reverseOrder (descending)oder: [practice, ide, geeks, contribute]
```

3)

```
package com.linkedhashsetdemo;

import java.util.LinkedHashSet;
public class LinkedHashSetDemo3 {
        public static void main(String[] args) {
                // Union : it will collect unique elements from the LinkedHashsets
```

```
        // Intersection (intersection is a type of set operation where the resultant set
        // will contain elements that are present in both of the sets.)
        // Difference
        LinkedHashSet<Integer> set1 = new LinkedHashSet<Integer>();
        set1.add(1);
        set1.add(3);
        set1.add(2);
        set1.add(5);
        set1.add(4);
        System.out.println("LinkedHash Set1:" + set1); // order is preserved
        LinkedHashSet<Integer> set2 = new LinkedHashSet<Integer>();
        set2.add(3);
        set2.add(4);
        set2.add(5);
        System.out.println("LinkedHash Set2:" + set2);
        // union: it will collect unique elements from the LinkedHashsets and give as a
        // result
        // set1.addAll(set2);
        // System.out.println("Union:" + set1);
        // intersection (common elements):the resultant set will contain elements that
        // are present in
        // both of the sets
        // set1.retainAll(set2);
        // System.out.println("intersection:" + set1);
        // Difference : it will remove common elements in LinkedHashset
        // set1.removeAll(set2);
        // System.out.println("Difference:" + set1);
        // subset: if set2 elements has present in set 1 then s2 is subset of s1
        // System.out.println("subset :"+ set1.containsAll(set2));
}
```

}

Output

1)

LinkedHash Set1:[1, 3, 2, 5, 4]

LinkedHash Set2:[3, 4, 5]

2)

LinkedHash Set1:[1, 3, 2, 5, 4]

LinkedHash Set2:[3, 4, 5]

Union:[1, 3, 2, 5, 4]

3)

LinkedHash Set1:[1, 3, 2, 5, 4]

LinkedHash Set2:[3, 4, 5]

intersection:[3, 5, 4]

4)

LinkedHash Set1:[1, 3, 2, 5, 4]

LinkedHash Set2:[3, 4, 5]

Difference:[1, 2]

5)

LinkedHash Set1:[1, 3, 2, 5, 4]

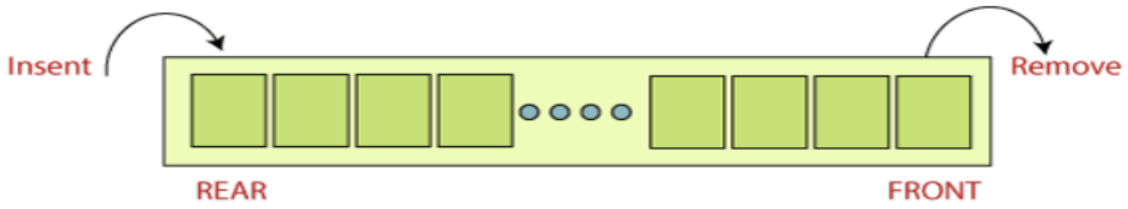
LinkedHash Set2:[3, 4, 5]

subset :true

Queue

A queue is another kind of linear data structure that is used to store elements just like any other data structure but in a particular manner. In simple words, we can say that the queue is a type of data structure in the Java programming language that stores elements of the same kind. The

components in a queue are stored in a FIFO (First In, First Out) behavior. There are two ends in the queue collection, i.e., front & rear. Queue has two ends that are front and rear.



The generic representation of the Java Queue interface is shown below:

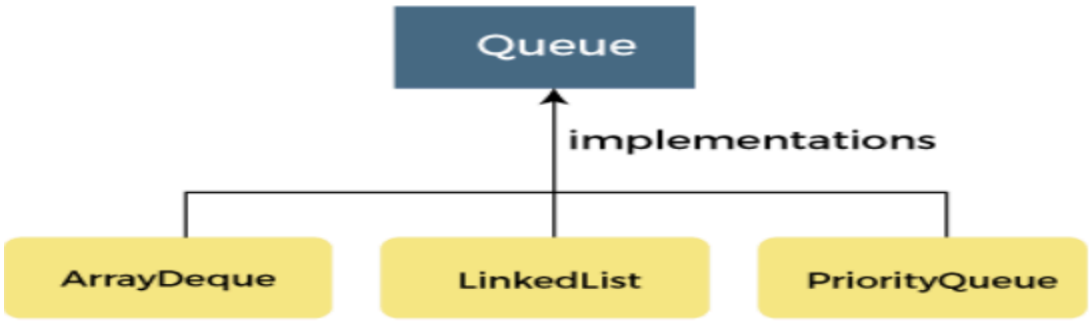
```
public interface Queue<T> extends Collection<T>
```

- Java Queue obeys the FIFO (First In, First Out) manner. It indicates that elements are entered in the queue at the end and eliminated from the front.
- The Java Queue interface gives all the rules and processes of the Collection interface like inclusion, deletion, etc.
- There are two different classes that are used to implement the Queue interface. These classes are LinkedList and PriorityQueue.
- Other than these two, there is a class that is, Array Blocking Queue that is used to implement the Queue interface.
- There are two types of queues, Unbounded queues and Bounded queues. The Queues that are a part of the java.util package are known as the Unbounded queues and bounded queues are the queues that are present in java.util.concurrent package.

- The Deque or (double-ended queue) is also a type of queue that carries the inclusion and deletion of elements from both ends.
- The deque is also considered thread-safe.
- Blocking Queues are also one of the types of queues that are also thread-safe. The Blocking Queues are used to implement the producer-consumer queries.
- Blocking Queues do not support null elements. In Blocking queues, if any work similar to null values is tried, then the NullPointerException is also thrown.

The classes that are used to implement the functionalities of the queue are given as follows:

- ArrayDeque
- LinkedList
- PriorityQueue



The major difference between linkedlist and priorityQueue is ,both homogeneous and heterogeneous allowed in linked list but priority can accept only homogeneous data.

Insertion order is allowed and duplicates are allowed in both the classes(linkedlist and priorityQueue).b

priorityQueue doesn't allow null value . and whereas linked lists allow as many null values.

Methods used in both the classes(linkedlist and priorityQueue).

add() vs offer()

i) **add() :** Adds elements to the queue at the end (tail) of the queue without violating the restrictions on the capacity. Returns true if success or return exception(IllegalStateException)if the capacity is exhausted or add is not successful.

ii) **offer() :** It is also used to add elements to the queue at the end . But The major difference between add() and offer() is, In offer method() If the insertion/ adding elements is successful then it returns true and if adding is not successful then return false.

element() vs peek()

iii) **element():** Returns the head (front) of the queue . Throws exception (NoSuchElementException) when the queue is empty.

iv) **peek():** Returns the head (front) of the queue as like the element() method. But it returns null value when the queue is empty.

remove() vs poll

v) **remove(): Removes the head of the queue and returns it.** Throws exception(NoSuchElementException) if the queue is empty.

vi) **poll():**Removes the head of the queue and returns it. If the queue is empty, it returns null.

1) **PriorityQueue**

```
package com.queuedemo;

import java.util.Iterator;
import java.util.PriorityQueue;
public class QueueDemo {
	/*
	 * The major difference between linkedList and priorityQueue is , both
	 * homogeneous and heterogeneous allowed in linked list but priority can accept
	 * only homogeneous data. Insertion order is allowed and duplicates are allowed
	 * in both the classes(linkedList and priorityQueue).
	 */
	public static void main(String[] args) {
		PriorityQueue pq = new PriorityQueue(); // It accepts only homogeneous data only
		// add :Adds elements to the queue at the end (tail) of the queue without
		// violating the restrictions on the capacity. Returns true if success or return
		// exception(IllegalStateException )if the capacity is exhausted or add is not
		// successful.
		pq.add("a");
		pq.add("b");
		pq.add("c");
		pq.add("c");
		//pq.add(null); null pointer exception
		// pq.add(100);not allowed
		System.out.println(pq); // insertion order is preserved and duplicates allowed.
```

```
// offer(): It is also used to add elements to the queue at the end . But The
// major difference between add() and offer() is, In offer method() If the
// insertion/ adding elements is successful then it returns true and if adding
// is not successful then return false.
pq.offer("d");
pq.offer("e");
pq.offer("f");
pq.offer("f");
System.out.println(pq); //// insertion order is preserved
// want to get head elements element() or peek()
// element(): Returns the head (front) of the queue . Throws exception
// (NoSuchElementException) when the queue is empty.
System.out.println(pq.element());
// peek() : Returns the head (front) of the queue as like the element() method.
// But it returns null value when the queue is empty.
System.out.println(pq.peek());
// return and remove the element remove() or poll()
// remove():Removes the head of the queue and returns it. Throws
// exception(NoSuchElementException) if the queue is empty.
System.out.println(pq.remove());
System.out.println("after remove operation" + pq);
// poll():Removes the head of the queue and returns it. If the queue is empty,
// it returns null.
System.out.println(pq.poll());
System.out.println("after poll operation" + pq);
```

```
                // isEmpty()
                System.out.println(pq.isEmpty());
                // contains
                System.out.println(pq.contains("e"));
                // Iterator()
                Iterator itr = pq.iterator();
                while (itr.hasNext()) {
                        System.out.println(itr.next());
                }
                // for each
                for (Object o : pq) {
                        System.out.println(o);
                }
        }
}
```

Output

```
[a, b, c, c]
[a, b, c, c, d, e, f, f]
a
a
a
after remove operation[b, c, c, f, d, e, f]
b
after poll operation[c, d, c, f, f, e]
```

false

true

c

d

c

f

f

e

c

d

c

f

f

e

2) linkedlist

```
package com.queuedemo;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.PriorityQueue;
public class QueueDemo2 {
	/*
	 * The major difference between linkedList and priorityQueue is , both
	 * homogeneous and heterogeneous allowed in linked list but priority can accept
	 * only homogeneous data. Insertion order is allowed and duplicates are allowed
```

```
     * in both the classes(linkedList and priorityQueue).
     */
    public static void main(String[] args) {
        LinkedList ll = new LinkedList(); // It accepts only both heterogeneous and homogeneous data only
        // add :Adds elements to the queue at the end (tail) of the queue without
        // violating the restrictions on the capacity. Returns true if success or return
        // exception(IllegalStateException )if the capacity is exhausted or add is not
        // successful.
        ll.add('A');
        ll.add("ram");
        ll.add(12);
        ll.add(3.56);
        ll.add(null);
        ll.add(null); //It accepts null values and we can add as many null values as you like..
        ll.add(true);
        System.out.println(ll); // insertion order is preserved and duplicates allowed.
        // offer(): It is also used to add elements to the queue at the end . But The
        // major difference between add() and offer() is, In offer method() If the
        // insertion/ adding elements is successful then it returns true and if adding
        // is not successful then return false.
        ll.offer("ram");
        ll.offer(1);
        ll.offer(8.09);
```

```
ll.offer(3.56);

ll.offer(false);

System.out.println(ll); //// insertion order is preserved

// want to get head elements element() or peek()

// element(): Returns the head (front) of the queue . Throws exception

// (NoSuchElementException) when the queue is empty.

System.out.println(ll.element());

// peek() : Returns the head (front) of the queue as like the element() method.

// But it returns null value when the queue is empty.

System.out.println(ll.peek());

// return and remove the element remove() or poll()

// remove():Removes the head of the queue and returns it. Throws

// exception(NoSuchElementException) if the queue is empty.

System.out.println(ll.remove());

System.out.println("after remove operation" + ll);

// poll():Removes the head of the queue and returns it. If the queue is empty,

// it returns null.

System.out.println(ll.poll());

System.out.println("after poll operation" + ll);

// isEmpty()

System.out.println(ll.isEmpty());

// contains

System.out.println(ll.contains("e"));

// Iterator()

Iterator itr = ll.iterator();
```

```
                while (itr.hasNext()) {
                        System.out.println(itr.next());
                }
                // for each
                for (Object o : ll) {
                        System.out.println(o);
                }
        }
}
```

Output

```
[A, ram, 12, 3.56, null, null, true]
[A, ram, 12, 3.56, null, null, true, ram, 1, 8.09, 3.56, false]
A
A
A
after remove operation[ram, 12, 3.56, null, null, true, ram, 1, 8.09, 3.56, false]
ram
after poll operation[12, 3.56, null, null, true, ram, 1, 8.09, 3.56, false]
false
false
12
3.56
null
null
```

true

ram

1

8.09

3.56

false

12

3.56

null

null

true

ram

1

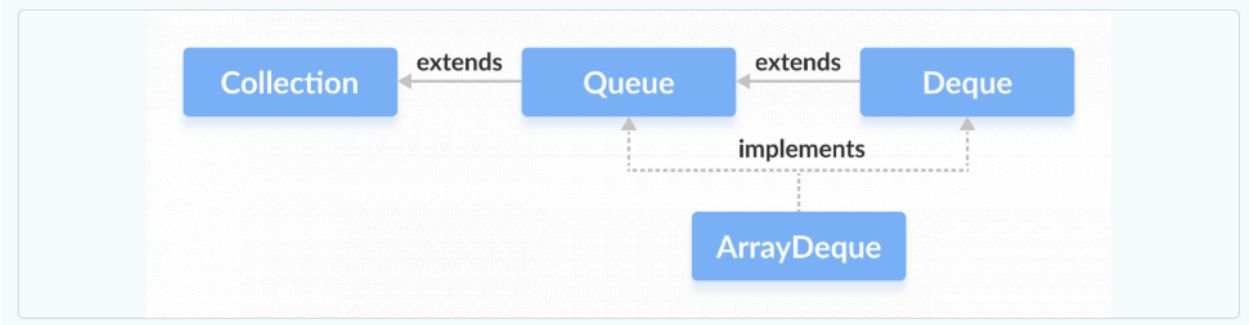
8.09

3.56

False

ArrayDeque

ArrayDeque class to implement queue and deque data structures using arrays.



ArrayDeque<Type> animal = new ArrayDeque<>();

ArrayDeque<String> animals = new ArrayDeque<>();

ArrayDeque<Integer> age = new ArrayDeque<>();

Add elements using add(), addFirst() and addLast()

add() - inserts the specified element at the end of the array deque

addFirst() - inserts the specified element at the beginning of the array deque

addLast() - inserts the specified at the end of the array deque (equivalent to add())

offer() - inserts the specified element at the end of the array deque

offerFirst() - inserts the specified element at the beginning of the array deque

offerLast() - inserts the specified element at the end of the array deque

getFirst() - returns the first element of the array deque

getLast() - returns the last element of the array deque

peek() - returns the first element of the array deque

peekFirst() - returns the first element of the array deque (equivalent to peek())

peekLast() - returns the last element of the array deque

remove() - returns and removes an element from the first element of the array deque

remove(element) - returns and removes the specified element from the head of the array deque

removeFirst() - returns and removes the first element from the array deque (equivalent to remove())

removeLast() - returns and removes the last element from the array deque

poll() - returns and removes the first element of the array deque

pollFirst() - returns and removes the first element of the array deque (equivalent to poll())

pollLast() - returns and removes the last element of the array deque

1)**package** com.arraydequedemo;

import java.util.ArrayDeque;

public class ArrayDequeDemo {

```
public static void main(String[] args) {
        ArrayDeque<String> animals = new ArrayDeque<>();
        // Using add()
        animals.add("Dog");
        // Using addFirst()
        animals.addFirst("Cat");
        // Using addLast()
        animals.addLast("Horse");
        System.out.println("ArrayDeque: " + animals);
        // Using offer()
        animals.offer("cow");
        // Using offerFirst()
        animals.offerFirst("Cat");
        // Using offerLast()
        animals.offerLast("elephant");
        System.out.println("ArrayDeque: " + animals);
        // Get the first element
        String firstElement = animals.getFirst();
        System.out.println("First Element: " + firstElement);
        // Get the last element
        String lastElement = animals.getLast();
        System.out.println("Last Element: " + lastElement);
        // Using peek()
        String element = animals.peek();
        System.out.println("Head Element: " + element);
```

```
// Using peekFirst()
String firstElement1 = animals.peekFirst();
System.out.println("First Element: " + firstElement1);
// Using peekLast
String lastElement1 = animals.peekLast();
System.out.println("Last Element: " + lastElement1);
// Using remove()
String element1 = animals.remove();
System.out.println("Removed Element: " + element1);
System.out.println("New ArrayDeque: " + animals);
// Using removeFirst()
String firstElement2 = animals.removeFirst();
System.out.println("Removed First Element: " + firstElement2);
// Using removeLast()
String lastElement2 = animals.removeLast();
System.out.println("Removed Last Element: " + lastElement2);
// Using poll()
String element2 = animals.poll();
System.out.println("Removed Element: " + element2);
System.out.println("New ArrayDeque: " + animals);
// Using pollFirst()
String firstElement3 = animals.pollFirst();
System.out.println("Removed First Element: " + firstElement3);
// Using pollLast()
String lastElement3 = animals.pollLast();
```

```
            System.out.println("Removed Last Element: " + lastElement3);

            // Using clear()

            animals.clear();

            System.out.println("New ArrayDeque: " + animals);

        }

}
```

Output

```
First Element: Cat

Last Element: elephant

Removed Element: Cat

New ArrayDeque: [Cat, Dog, Horse, cow, elephant]

Removed First Element: Cat

Removed Last Element: elephant

Removed Element: Dog

New ArrayDeque: [Horse, cow]

Removed First Element: Horse

Removed Last Element: cow

New ArrayDeque: []
```

ArrayDeque Vs. LinkedList Class

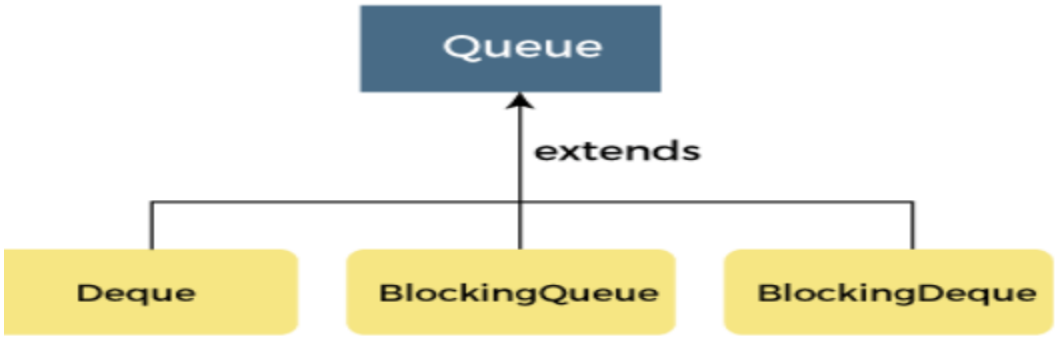
Both ArrayDeque and Java LinkedList implements the Deque interface. However, there exist some differences between them.

LinkedList supports null elements, whereas ArrayDeque doesn't.

Each node in a linked list includes links to other nodes. That's why LinkedList requires more storage than ArrayDeque.

If you are implementing the queue or the deque data structure, an ArrayDeque is likely to faster than a LinkedList.

Interfaces which extend the queue.



Deque:

The Deque interface of the Java collections framework provides the functionality of a double-ended queue. It extends the Queue interface.

In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements from both front and rear.

Both ArrayDeque and LinkedList classes implement the Deque interface.

// Array implementation of Deque

Deque<String> animal1 = new ArrayDeque<>();

// LinkedList implementation of Deque

Deque<String> animal2 = new LinkedList<>()

Since Deque extends the Queue interface, it inherits all the methods of the Queue interface.

Whatever methods available in arraydeque the same methods are available here.

Map

Map Interface

The scenario where we need to represent group data as a key-value pair then will use map concepts.

It is not a child interface of collection interface.
Here the data/object is in the form of a key-value pair.
The **keys** are unique and not duplicate.
The **values** can be duplicated.

Classes which implement Map Interface
i)HashMap ii)HashTable iii)LinkedHashMap

HashMap

Insertion is not preserved as it uses hashcode methodology.
Duplicates keys are not allowed and it should be unique.
Duplicate values are allowed.
We can store the key as a "null" one time.
We can add value as a null as many null values as you .
We can suggest HasMap when frequent search operations are involved.

Declaration

HashMap hs = new HashMap(); //stores heterogenous data.
Map<String, Integer> phoneBook = new HashMap<String, Integer>(); // store homogenous data.
HashMap<Integer, String> hm = new HashMap<Integer, Integer>();
public class HashMap<K,V> extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

Methods:
put(K key, V value): Adds a key-value pair to the HashMap. If the key already exists, the old value is replaced with the new value.

putAll(Map m) : method copies all key-value pairs from the specified map m into the current HashMap.

get(Object key): Retrieves the value associated with the specified key. Returns null if the key does not exist.
remove(Object key): Removes the key-value pair associated with the specified key. Returns the value that was associated with the key, or null if the key was not found.
containsKey(Object key): return true if the HashMap contains a mapping for the specified key else return false.
containsvalue(value): return true if the HashMap contains a mapping for the specified value ,else return false .
size(): Returns the number of key-value pairs in the HashMap.
clear(): Removes all key-value pairs from the HashMap.
isEmpty(): checks whether hashmap is empty or not.
keySet(): It will return keys as set.
keyValues(): It will return values as collection
entrySet(): It will return all entry sets as a one set. method returns a set view of all the entries.

Entry Interface

The interface which is created inside HashMap.
Map.Entry interface is a nested interface within the Map interface. It represents a key-value pair in a map. Each entry in a Map is an instance of this interface, and it provides methods to access and manipulate the key and value associated with the entry.
Methods only for Entry interface:
getKey():

- Description: Returns the key associated with this entry.
- Return Type: K.
  K key = entry.getKey();

getValue():

- Description: Returns the value associated with this entry.
- Return Type: V.
  V value = entry.getValue();

setValue(V value):

- Description: Replaces the value associated with this entry with the specified value.
- Return Type: V (the old value).
  V oldValue = entry.setValue("newValue");

Program

```
package com.hashmapdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
public class HashMapDemo {
	public static void main(String[] args) {
		HashMap hm = new HashMap(); // Homogeneous data
		// HashMap<Integer, String> hm = new HashMap<Integer, Integer>();
		hm.put(101, "ram");
		hm.put(102, "dhoni");
		hm.put(103, "rohit");
		hm.put(104, "rahul");
		hm.put(105, "ram");// value can be duplicated
		hm.put(106, "raju");
		hm.put(null, "virat"); // only one key can should be null value
		hm.put("dhoni", "ms");
		// hm.put(101, "ram"); key can't be duplicated , if we add duplicate key it will
		// replace exiting value of that key.
		System.out.println(hm); // order is not preserved.
		// size
		System.out.println("size of hashmap : " + hm.size());
		// get()
		System.out.println(hm.get(101));
		// remove pair of hashmap
		System.out.println(hm.remove("dhoni"));
		System.out.println(hm.remove(101));
		System.out.println(hm);
```

```
// containsKey
System.out.println(hm.containsKey(106));// true
System.out.println(hm.containsKey(101));// false
// ContainsValue
System.out.println(hm.containsValue("rohit"));// true
System.out.println(hm.containsValue("harsh"));// false
// isEmpty
System.out.println(hm.isEmpty());
// KeySet() : return all keys as set(as if we want return key as set it shoudn't
// contain duplicate so that's why result is in set)
System.out.println(hm.keySet());
// values(): return all values as a collection(because values can be duplicated,
// so that's why result is in collection)
System.out.println(hm.values());
// entrySet(): return all entries(key-value pair) as a set
System.out.println(hm.entrySet());
// entry interface.
// it will read all keys individually.
for (Object o : hm.keySet()) {
	System.out.println(o);
}
// we can also get all the values individually
for (Object o1 : hm.values()) {
	System.out.println(o1);
}
// we can also read all keys individually with respective values. Using get
// method we can get values from respective keys.
for (Object o3 : hm.keySet()) {
	System.out.println(o3 + " " + hm.get(o3));
}
// Entry interface specific methods
// **********************************
System.out.println("Entry Interface methods");
HashMap<Integer, String> m = new HashMap<Integer, String>();
m.put(101, "ram");
m.put(102, "dhoni");
m.put(103, "rohit");
m.put(104, "rahul");
m.put(105, "ram");// value can be duplicated
m.put(106, "raju");
```

```
                m.put(null, "virat");
                for (Map.Entry entry : m.entrySet()) {
                        System.out.println(entry.getKey() + " " + entry.getValue());
                }
                // iterator() method
                System.out.println("itertor() method in Map.Entry interface");
                Set s = m.entrySet(); // the result of entryset is set
                Iterator itr = s.iterator();
                while (itr.hasNext()) {
                        Map.Entry entry = (Entry) itr.next();
                        System.out.println(entry.getKey() + " " + entry.getValue());
                }
        }
}
```

Output

```
{null=virat, dhoni=ms, 101=ram, 102=dhoni, 103=rohit, 104=rahul, 105=ram, 106=raju}
size of hashmap : 8
ram
ms
ram
{null=virat, 102=dhoni, 103=rohit, 104=rahul, 105=ram, 106=raju}
true
false
true
false
false
[null, 102, 103, 104, 105, 106]
[virat, dhoni, rohit, rahul, ram, raju]
[null=virat, 102=dhoni, 103=rohit, 104=rahul, 105=ram, 106=raju]
null
102
103
104
105
106
virat
dhoni
rohit
rahul
ram
```

raju
null virat
102 dhoni
103 rohit
104 rahul
105 ram
106 raju
Entry Interface methods
null virat
101 ram
102 dhoni
103 rohit
104 rahul
105 ram
106 raju
itertor() method in Map.Entry interface
null virat
101 ram
102 dhoni
103 rohit
104 rahul
105 ram
106 raju

HashTable

The Hash table data structure stores elements in key-value pairs where

- Key- unique integer that is used for indexing the values
- Value - data that are associated with keys.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Whatever the methods available in Map interface .The same methods are available here.
The major difference between hashmap and hashtable.

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

But there are many differences between HashMap and Hashtable classes that are given below.

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**. Duplicate values are allowed. The order is not preserved.	Hashtable **doesn't allow any null value or key**.Duplicate values are allowed.The order is not preserved.
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.

```
package com.hashtabledemo;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;
public class HashtableDemo {
	public static void main(String[] args) {
		// Hashtable t = new Hashtable();The initial default capacity of Hashtable class
		// is 11 whereas loadFactor is 0.75.
		// int initialCapacity=89;
		// Hashtable t1 = new Hashtable(initialCapacity);
		// Hashtable t2 = new Hashtable(initialCapacity,fillratio/loadfactor);
```

```
// Hashtable<String, Integer> hashtable = new Hashtable<>();
// Hashtable<String, Integer> hashtable = new Hashtable<>(20); // 20 is the
// initial capacity
// Hashtable<String, Integer> hashtable = new Hashtable<>(20, 0.75f); // 20 is
// the initial capacity, 0.75 is the load factor
Hashtable<Integer, String> ht = new Hashtable<Integer, String>();
ht.put(101, "dhoni");
ht.put(102, "ms");
ht.put(103, "virat");
ht.put(104, "dhoni"); // value can be duplicated
ht.put(105, "rahul");
// ht.put(101,"null"); won't allow will get NullPointerException
// ht.put(null,"dhoni"); key shoudn't be null here
System.out.println(ht);
// size
System.out.println("size of hashmap : " + ht.size());
// get()
System.out.println(ht.get(105));
// remove pair of hashmap
System.out.println(ht.remove(101));
System.out.println(ht);
// containsKey
System.out.println(ht.containsKey(104));// true
System.out.println(ht.containsKey(101));// false
// ContainsValue
System.out.println(ht.containsValue("virat"));// true
System.out.println(ht.containsValue("harsh"));// false
// isEmpty
System.out.println(ht.isEmpty());
// KeySet() : return all keys as set(as if we want return key as set it shoudn't
// contain duplicate so that's why result is in set)
System.out.println("return all keys as set");
System.out.println(ht.keySet());
// values(): return all values as a collection(because values can be duplicated,
// so that's why result is in collection)
System.out.println("return all values as a collection");
System.out.println(ht.values());
// entrySet(): return all entries(key-value pair) as a set
System.out.println("return all entries(key-value pair) as a set");
System.out.println(ht.entrySet());
```

```
                // entry interface.
                // it will read all keys individually.
                System.out.println("reading all keys individually");
                for (Object o : ht.keySet()) {
                        System.out.println(o);
                }
                System.out.println("reading all values individually");
                // we can also get all the values individually
                for (Object o1 : ht.values()) {
                        System.out.println(o1);
                }
                // we can also read all keys individually with respective values. Using get
                // method we can get values from respective keys.
                System.out.println("reading keys along respective values");
                for (int i : ht.keySet()) {
                        System.out.println(i + " " + ht.get(i));
                }
                // Entry interface specific methods
                // **********************************
                System.out.println("Map.Entry method");
                for (Map.Entry entry : ht.entrySet()) {
                        System.out.println(entry.getKey() + " " + entry.getValue());
                }
                // iterator() method
                System.out.println("itertor() method in Map.Entry interface");
                Set s = ht.entrySet(); // the result of entryset is set
                Iterator itr = s.iterator();
                while (itr.hasNext()) {
                        Map.Entry entry = (Entry) itr.next();
                        System.out.println(entry.getKey() + " " + entry.getValue());
                }
        }
}
```

Output

```
{105=rahul, 104=dhoni, 103=virat, 102=ms, 101=dhoni}
size of hashmap : 5
rahul
dhoni
{105=rahul, 104=dhoni, 103=virat, 102=ms}
true
```

false
true
false
false
return all keys as set
[105, 104, 103, 102]
return all values as a collection
[rahul, dhoni, virat, ms]
return all entries(key-value pair) as a set
[105=rahul, 104=dhoni, 103=virat, 102=ms]
reading all keys individually
105
104
103
102
reading all values individually
rahul
dhoni
virat
ms
reading keys along respective values
105 rahul
104 dhoni
103 virat
102 ms
Map.Entry method
105 rahul
104 dhoni
103 virat
102 ms
itertor() method in Map.Entry interface
105 rahul
104 dhoni
103 virat
102 ms

LinkedHashMap

The **LinkedHashMap Class** is just like HashMap with an additional feature of maintaining an order of elements inserted into it. The only difference from hashmap is it preserves order.

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

The same methods of hashmap can be implemented here.

i) LinkedHashMap map = new LinkedHashMap();The initial default capacity ofLinkedHashMap class is 16 whereas loadFactor is 0.75.

ii) int initialCapacity=89;

LinkedHashMap t1 = new LinkedHashMap(initialCapacity);

iii) LinkedHashMap t2 = new LinkedHashMap(initialCapacity, fill ratio/load factor);

iv) a) Create a LinkedHashMap with accessOrder = true

boolean accessOrder = true;

LinkedHashMap<String, Integer> map = newLinkedHashMap<>(20, 0.75f, accessOrder);//here insertion order is not Persevered.

b) Create a LinkedHashMap with accessOrder = false boolean

accessOrder = false;

LinkedHashMap<String, Integer> map = new

LinkedHashMap<>(20, 0.75f, accessOrder);(here insertion order is persevered).

programm:

```
package com.linkedhashmapdemo;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;
public class LinkedHashMapDemo {
	public static void main(String[] args) {
		// LinkedHashMap map = new LinkedHashMap();The initial default capacity of
		// LinkedHashMap class
		// is 16 whereas loadFactor is 0.75.
		// int initialCapacity=89;
		// LinkedHashMap t1 = new LinkedHashMap(initialCapacity);
		// LinkedHashMap t2 = new LinkedHashMap(initialCapacity,fillratio/loadfactor);
		// Create a LinkedHashMap with accessOrder = true
		/*
		 * boolean accessOrder = true;
		 * LinkedHashMap<String, Integer> map = newLinkedHashMap<>(20, 0.75f,
accessOrder);(here insertion order is not
		 * Persevered).
		 * // Create a LinkedHashMap with accessOrder = false boolean
		 * accessOrder = false;
		 * LinkedHashMap<String, Integer> map = new
		 * LinkedHashMap<>(20, 0.75f, accessOrder);(here insertion order is
persevered).
```

```
 */
// LinkedHashMap<String, Integer> LinkedHashMap = new
LinkedHashMap<>();
// LinkedHashMap<String, Integer> LinkedHashMap = new
LinkedHashMap<>(20); // 20
// is the
// initial capacity
// LinkedHashMap<String, Integer> LinkedHashMap = new
LinkedHashMap<>(20,
// 0.75f); // 20 is
// the initial capacity, 0.75 is the load factor
LinkedHashMap<Integer, String> lh = new LinkedHashMap<Integer, String>();
lh.put(101, "dhoni");
lh.put(102, "ms");
lh.put(103, "virat");
lh.put(104, "dhoni"); // value can be duplicated
lh.put(105, "rahul");
lh.put(106, "null"); // value can null as many as
lh.put(107, "null");
// lh.put(null,"dhoni"); key shoudn't be null here
System.out.println(lh);
// size
System.out.println("size of hashmap : " + lh.size());
// get()
System.out.println(lh.get(105));
```

```
// remove pair of LinkedHashMap
System.out.println(lh.remove(101));
System.out.println(lh);
// containsKey
System.out.println(lh.containsKey(104));// true
System.out.println(lh.containsKey(101));// false
// ContainsValue
System.out.println(lh.containsValue("virat"));// true
System.out.println(lh.containsValue("harsh"));// false
// isEmpty
System.out.println(lh.isEmpty());
// KeySet() : return all keys as set(as if we want return key as set it shoudn't
// contain duplicate so that's why result is in set)
System.out.println("return all keys as set");
System.out.println(lh.keySet());
// values(): return all values as a collection(because values can be duplicated,
// so that's why result is in collection)
System.out.println("return all values as a collection");
System.out.println(lh.values());
// entrySet(): return all entries(key-value pair) as a set
System.out.println("return all entries(key-value pair) as a set");
System.out.println(lh.entrySet());
// entry interface.
// it will read all keys individually.
System.out.println("reading all keys individually");
```

```
for (Object o : lh.keySet()) {
        System.out.println(o);
}
System.out.println("reading all values individually");
// we can also get all the values individually
for (Object o1 : lh.values()) {
        System.out.println(o1);
}
// we can also read all keys individually with respective values. Using get
// method we can get values from respective keys.
System.out.println("reading keys along respective values");
for (int i : lh.keySet()) {
        System.out.println(i + " " + lh.get(i));
}
// Entry interface specific methods
// **********************************
System.out.println("Map.Entry method");
for (Map.Entry entry : lh.entrySet()) {
        System.out.println(entry.getKey() + " " + entry.getValue());
}
// iterator() method
System.out.println("itertor() method in Map.Entry interface");
Set s = lh.entrySet(); // the result of entryset is set
Iterator itr = s.iterator();
while (itr.hasNext()) {
```

```
                Map.Entry entry = (Entry) itr.next();
                System.out.println(entry.getKey() + " " + entry.getValue());
            }
        }
}
```

Output

```
{101=dhoni, 102=ms, 103=virat, 104=dhoni, 105=rahul, 106=null, 107=null}
size of hashmap : 7
rahul
dhoni
{102=ms, 103=virat, 104=dhoni, 105=rahul, 106=null, 107=null}
true
false
true
false
false
return all keys as set
[102, 103, 104, 105, 106, 107]
return all values as a collection
[ms, virat, dhoni, rahul, null, null]
return all entries(key-value pair) as a set
[102=ms, 103=virat, 104=dhoni, 105=rahul, 106=null, 107=null]
reading all keys individually
102
```

103

104

105

106

107

reading all values individually

ms

virat

dhoni

rahul

null

null

reading keys along respective values

102 ms

103 virat

104 dhoni

105 rahul

106 null

107 null

Map.Entry method

102 ms

103 virat

104 dhoni

105 rahul

106 null

107 null

itertor() method in Map.Entry interface

102 ms

103 virat

104 dhoni

105 rahul

106 null

107 null