Collection Framework

A COMPLETE GUIDE IN SIMPLE WAY

By-Avinash Pingale
TESTING SHASTRA | WWW.TESTINGSHASTRA.COM | +91-9130502135

Collection Framework

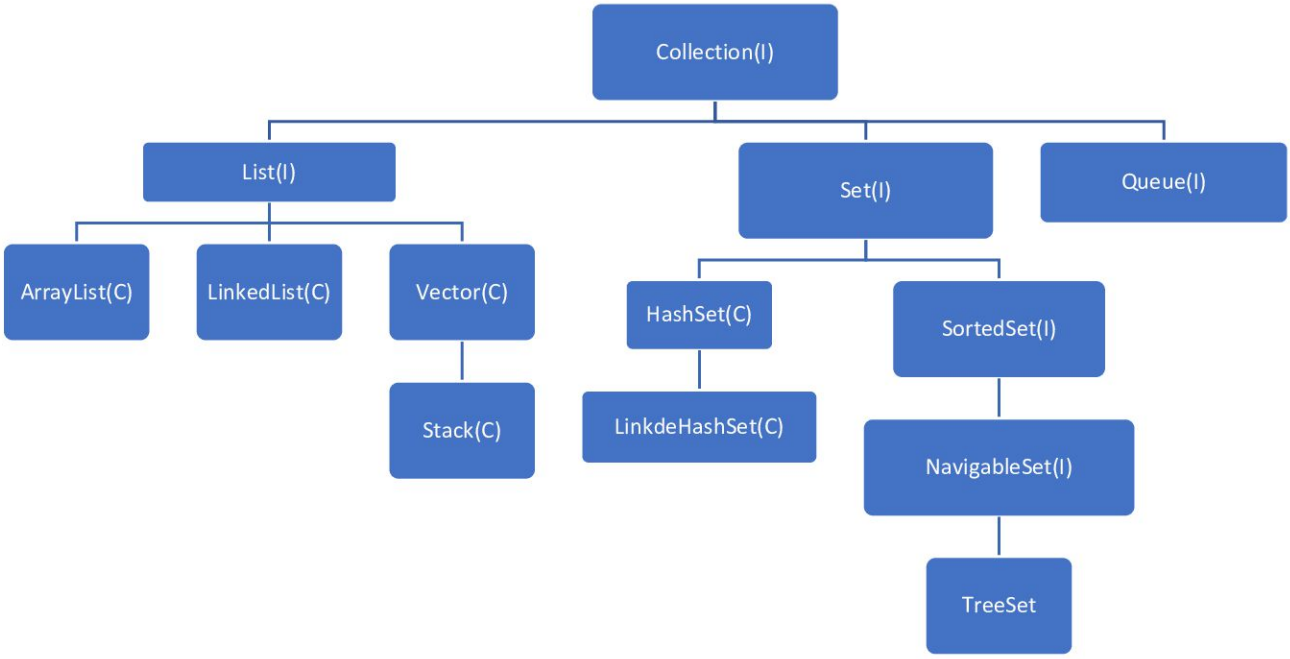
- To store single constant value, we can use variables.
- For e.g to store 10 in memory, we can use int i=10;
- To store multiple constant values, we can use arrays.
- E.g. int[] x={1,2,3,4,5};
- But what if we want to store multiple objects?
- Then we can obviously create an Object array. It seems heterogenous but it is not. Because it will automatically convert all elements (new Student(), new Employee(), new String("Hello") and 10) to Object class instance by rule of implicit type casting.
- Hence all instances will loose their original property. For e.g we can increment 10, but in below example we cannot write: obj[3]++;
- E.g. Object[] obj=new Object[4];
  obj[0]=new Student();
  obj[1]=new Employee();
  obj[2]=new String("Hello");
  obj[3]=10;
- Arrays have certain limitations:
  - Arrays are fixed in size. Their size cannot be changed at runtime.
  - We cannot create heterogenous array who preserves original property of objects.
  - Readymade method support is not available in array for operations like: add, remove, search, sort etc.
  - Only one data structure is supported, that is array.
- Hence Java have provided Collection Framework to rectify all problems of Array.

Properties of Collection Framework:

- Collection can store multiple objects.
- Collection can store only Non-primitive data elements. If we try to insert primitive data elements, then we will not get any error. Whereas primitive data elements will automatically be converted into Non-primitive type object.
- It can store homogenous as well as heterogenous objects.
- Collection is growable in nature.
- Collection have readymade method support for operations like: add, remove, search, sort etc.
- Collection provides different data structures to store objects, like: Array, Stack, Queue, PriorityQueue, Set, Tree etc.

Hierarchy of Collection Framework:

On NEXT PAGE



1. **Collection(I):**
   - It is a parent interface of Collection framework.
   - When we want to represent group of objects which can grow at runtime, then we will use Collection.
   - It has methods, which are implemented by almost all classes of Collection Framework.
   - It has three child interfaces: List, Set and Queue.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                ArrayList al=new ArrayList();
                al.add("Welcome");
                al.add("To");
                al.add("Testing");
                al.add("Shastra");
                al.add("This");
                al.add("is");
                al.add("Avinash");
        }
}
```

Above example will be considered for all methods and their description

Methods of Collection interface:

1. **int Size():**
   a. It returns number of elements present inside this Collection.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                Syste.out.println("Number of Elements: "+al.size())
        }
}
```

2. **boolean isEmpty():**
    a. It will return true iff this collection has no elements in it.
    b. It will return false if this collection has atleast one element.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                System.out.println("is Empty? "+al.isEmpty())
        }
}
```

    c. In above example, it will print 'false'

3. **boolean contains(Object o):**
    a. It will search the given object inside the collection.
    b. It will return true if specified element is present in this collection.
    c. It will return false if the specified element is absent in this collection.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                System.out.println("Search Testing"+al.contains("Testing"))
        }
}
```

    d. It will return `true` because "Testing" is present inside `al`.

4. **Object[] toArray():**
    a. It will return Object array representation of the Collection.
    b. In short, it will convert any collection to an Object array.
    c. If 'al' contains heterogenous data elements, then we cannot cast it to String[] array as shown in in below example.
    d. For that, we need to compulsorily create an Object[] array.

```
public class ArrayListDemo {
        public static void main(String[] args) {
          String[] elements=(String[])al.toArray();
        }
}
```

5. **boolean add(Object o):**
    a. It will insert the given element at the last position of the collection.
    b. It will return true if element is successfully added inside the collection.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                al.add("We are in Pune");
                System.out.println(al)
        }
}
```

    c. "We are in Pune" text will be appended at the last index of an ArrayList al.

6. **boolean remove(Object o):**
    a. It removes the specified element from the collection.

b. Provided Object o should be present inside this collection.
c. If Object o is not present in this collection, then it will return false.
d. E.g.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                al.remove("We are in Pune")
                System.out.println(al)
        }
}
```

7. **boolean containsAll(Collection c):**
   a. It will search all matching elements from Collection c in this collection.
   b. E.g.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                LinkedList l=new LinkedList();
                l.add("Testing");
                l.add("Shastra");

                boolean flag=al.containsAll(l);
                System.out.println(flag);
        }
}
```

8. boolean addAll(Collection c):
   a. It will append all elements from Collection c to the last index of this collection.
   b. E.g.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                LinkedList l=new LinkedList();
                l.add("Testing");
                l.add("Shastra");

                al.addAll(l);
                System.out.println(al);
        }
}
```

   c. Above program will return updated al, where we can see duplicate "Testing" and "Shastra".
   d. If this collection is of type Set, then `addAll()` method will return `'false'` as values are going to be duplicate.
9. **boolean removeAll(Collection c):**
   a. It will remove all elements from this collection which are exact matching with elements from Collection c.
   b. In short, this method will remove common elements from this collection.
   c. If there are no common elements to remove, then it will return false.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                LinkedList l=new LinkedList();
                l.add("Testing");
                l.add("Shastra");

                al.removeAll(l);
                System.out.println(al);//["Welcome", "To", "This", "is", "Avinash"]
        }
}
```

10. **boolean retainAll(Collection c):**
    a. It will remove uncommon elements (between this collection and Collection c), from this collection.
    b. E.g.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                LinkedList l=new LinkedList();
                l.add("Testing");
                l.add("Shastra");

                al.retainAll(l);
                System.out.println(al);//["Testing", "Shastra"]
        }
}
```

List (I):

- List can store elements sequentially.
- Duplicates are allowed.
- Null insertion is possible.
- Insertion order is preserved.
- Both homogenous as well as heterogenous elements can be inserted.

Methods of List Interface:

1. boolean add(int index, Object o):
    a. It will add an Object o at specified index.
    b. If there is already an element at specified index, then that element will be shifted to right by one position.
    c. E.g.

```
public class ListMethodsDemo {
        public static void main(String[] args) {
                ArrayList al=new ArrayList();
                al.add(10);
                al.add(11);
                al.add(12);
                al.add(13);
                al.add(14);

                al.add(2, 17);

        }

}
```

d. In above example, 17 will be added to 2nd position and all subsequent elements will be shifted to right by 1 position.

2. boolean addAll(int index, Collection c):

a. It will copy all elements from Collection c in this collection at specified index.

b. If there is already some element at specified 'index' then all subsequent elements will be shifted to right to accommodate all elements from Collection c.

c. E.g.

```
public class ListMethodsDemo {
        public static void main(String[] args) {
                ArrayList al=new ArrayList();
                al.add(10);
                al.add(11);
                al.add(12);
                al.add(13);
                al.add(14);

                LinkedList link=new LinkedList();
                link.add(34);
                link.add(35);
                al.addAll(2, link);
        }
}
```

d. In above example, 34 and 35 will be added to 'al' at second position. 12,13,14 will be shifted to right to accommodate 34 and 35.

3. Object get(int index):

a. It will return an Object from this List located at specified index.

b. E.g.

```
public class ListMethodsDemo {
	public static void main(String[] args) {
		ArrayList al=new ArrayList();
		al.add(10);
		al.add(11);
		al.add(12);
		al.add(13);
		al.add(14);

		System.out.println(al.get(3));// It will print 13

	}
}
```

c. Above program will return 13.

4. Object set(int index, Object o):
   a. It will insert an Object o at specified index.
   b. If there is already some element at that index, then that element will be returned and new value will be set().
   c. E.g.

```
public class ListMethodsDemo {
	public static void main(String[] args) {
		ArrayList al=new ArrayList();
		al.add(10);
		al.add(11);
		al.add(12);
		al.add(13);
		al.add(14);
		int oldValue=(int) al.set(2, 72);
		System.out.println(al);
	}
}
```

d. In above example, 12 will be replaced with 72. And 12 will be returned by set() method and it will be saved in 'oldValue' variable.

5. Object remove(int index):
   a. It will remove the element at specified index and return that element.
   b. If there are elements present at the right of the removed element, then all those subsequent elements will be shifted to left.
   c. E.g.

```
public class ListMethodsDemo {
	public static void main(String[] args) {
		ArrayList al=new ArrayList();
		al.add(10);
		al.add(11);
		al.add(12);
		al.add(13);
		al.add(14);
		System.out.println("Before Remove: \n"+al);
		al.remove(2);
		System.out.println("After Remove: \n"+al);

	}
}
```

d. In above example, 12 will be removed. 13 and 14 will be shifted to left by one position.

6. int indexOf(Object o):
   a. It returns the position of Object o in this list.
   b. If there are duplicate elements in this list, then it returns the first occurrence of the element.
   c. E.g

```
public class ListMethodsDemo {
        public static void main(String[] args) {
                ArrayList al=new ArrayList();
                al.add(10);
                al.add(11);
                al.add(12);
                al.add(13);
                al.add(14);
                al.add(12);
                System.out.println(al.indexOf(12));

        }
}
```

   d. In above program, it will return 2 because 12 first occurs at 2nd index.

7. int lastIndexOf(Object o):
   a. It will return the last occurrence of the Object o.
   b. If there are duplicate elements, then it will return the index of last element.
   c. E.g.

```
public class ListMethodsDemo {
        public static void main(String[] args) {
                ArrayList al=new ArrayList();
                al.add(10);
                al.add(11);
                al.add(12);
                al.add(13);
                al.add(14);
                al.add(12);
                System.out.println(al.lastIndexOf(12));

        }
}
```

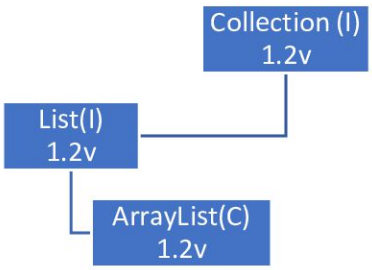
   d. In above program, it will return 5.

8. List<E> subList(int fromIndex, int lastIndex):
   a. It is used to create a sub-list from this List.
   b. Sublist will contain elements inclusive fromIndex and eclusive toIndex.
   c. E.g.

```
public class ListMethodsDemo {
	public static void main(String[] args) {
		ArrayList al=new ArrayList();
		al.add(10);
		al.add(11);
		al.add(12);
		al.add(13);
		al.add(14);
		al.add(12);
		ArrayList subList=(ArrayList) al.subList(1, 4);
		System.out.println(subList);

	}
}
```

d. In above program, subList will contain [11,12,13].

ArrayList(C):



- It backend data structure is Array.
- It can save homogenous as well as heterogenous data elements.
- It is growable in nature
- Duplicate elements are allowed
- Insertion order is preserved
- Null insertion is possible.
- Random access is possible, because it implements 'RandomAccess' interface.
- Its default capacity is 10.
- It grows when it reaches to maximum capacity. Growth factor is calculated using below formula:

NewCapacity=CurrentCapacity*3/2+1

Constructors of ArrayList():

1. **ArrayList():**
   a. It is used to construct ArrayList of capacity 10.
2. **ArrayList(int initialCapacity):**
   a. It is used to construct an ArrayList of specified initial capacity.
   b. This capacity can further be growable.
   c. Capacity should not be negative. If negative capacity is mentioned, then it will throw R.E: IllegalArgumentException
3. **ArrayList(Collection c):**

a. It is used to convert any Collection into an ArrayList.
b. It cannot be used to convert Map into an ArrayList.

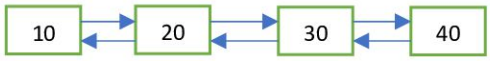
It implements all methods of Collection and List interface.

LinkedList(C) :

- Its backend data structure is doubly-linked-list.
- It can store homogenous as well as heterogenous data elements.
- It is growable in nature.
- Duplicates are allowed
- Insertion order is preserved
- Null insertion is possible.
- Random access is not possible.
- It doesn't have initial capacity.

Constructors of LinkedList:

1. **LinkedList():**
   a. It is used to create an empty LinkedList having no initial capacity.
2. **LinkedList(Collection c):**
   a. It is used to create an LinkedList having all elements from Collection c.
   b. In short, it is used to convert any collection into a LinkedList.

Sr.No	ArrayList	LinkedList
1	Backend data structure is growable array	Backend data structure is Doubly Linked List
2	Default capacity is 10	It has no default capacity
3	Data is stored continuously E.g. 10 | 20 | 30 | 40 | 50 | 60	Data is not stored continuously 
4	Random access is possible	Random access is not possible
5	Best if frequent operation is reading	Best ff frequent operation is insertion-deletion
6	Worst if frequent operations are insertion and deletion	Worst if frequent operations are reading

Methods of LinkedList:

Vector(C):

- Its backend data structure is Array.
- It introduced in 1.0 version of java
- It is called as legacy class of java.
- It can store homogenous as well as heterogenous data elements.
- Duplicates are allowed
- Insertion order is preserved.
- Null insertion is possible.
- Random access is possible, because it implements RandomAccess interface
- Vector is thread-safe. That is, Vector objects are synchronized.
- All methods of Vector are synchronized.

- It grows once it reaches to its capacity by double.

NewCapcity=CurrentCapacity*2

Difference between ArrayList and Vector:

Sr.No	ArrayList	Vector
1	It is introduced in 1.2v of java	It is introduced in 1.0v of Java
2	Non-legacy class	Legacy class
3	Backend data structure is Array	Backend data structure is Array
4	Growth ratio: NewCap=CurrentCap*3/2+1	Growth Ratio: NewCap=CurrentCap*2
5	Initial capacity is 10	Initial capacity is 10
6	ArrayList is not thread-safe	Vector is thread-safe
7	None of the methods of ArrayList are synchronized	All methods of Vector are synchronized
8	Performance is high	Performance is low
9	Not recommended in Multi-threaded environment	Recommended in Multi-threaded environment

Constructors of Vector:

1. **Vector():**
   a. It creates an empty vector of initial capacity: 10
   b. It can be growable after 10^{th} position.
2. **Vector(int initial Capacity):**
   a. It creates an empty Vector of specified initial capacity
3. **Vector(int intialCapacity, int increementalCapacity):**
   a. It is used to create an empty Vector of specified initial capacity
   b. Each time it reaches to max capacity, it will grow by specified incrementCapacity.
   c. E.g.

```
public class VectorDemo {
        public static void main(String[] args) {
                        Vector al=new Vector(5,3);
                        al.add(10);
                        al.add(20);
                        al.add(30);
                        al.add(40);
                        al.add(50);
                        al.add(60);
                        System.out.println(al.capacity());
        }
}
```

4. **Vector(Collection c):**
   a. It is used to convert any Collection c into a Vector.
   b. Newly created Vector will contain all elements from Collection c.

c. If Collection c is null, it will throw NullPointerException.

Methods of Vector:

1. addElement(Object o):
    a. It appends specified Object o at the last vacant position of the Vector.
    b. If Vector is empty, then it will add element at the 0^{th} index.
2. removeElementAt(int index):
    a. It removes an element from specified index.
    b. It will shift all subsequent elements to the left by one position.
3. removeElement(Object o):
    a. It will remove first occurrence of the Object o from vector.
    b. It will shift subsequent elements to the left by one position.
4. removeAllElements():
    a. It will remove all elements from the vector.
5. Object elementAt(int index):
    a. It returns element at specified index from this vector.
    b. Object firstElement():
    c. It returns first element of vector.
6. Object lastElement():
    a. It returns the last element from the vector.
7. int capacity():
    a. It returns the capacity of the vector. (Remember size and capacity are two different terms.)
8. Enumeration elements():
    a. It is used to iterate through given vector.

Stack:

- Stack is Last-In-First-Out Data structure.
- When we want to store data elements temporarily, then we will go for stack.
- It preserves insertion order
- Homogenous as well as Heterogenous objects are allowed
- Null insertion is possible
- Duplicate elements are allowed.
- It is introduced in 1.0v of java.
- Stack is implemented using array.

Constructors of Stack:

1. Stack():
    a. It creates an empty stack of no default initial capacity.

Methods of Stack():

1. Object push(Object o):
    a. It inserts an Object o at the top of the stack and increments the offset.
    b. It returns the same object which it is adding to the stack.
    c. It is not synchronized operation. In short it is not thread-safe.
2. Object pop():

a. It will return the element present at the top of the stack
b. It also removes that element from the top and updates the top.
c. pop() method is synchronized in stack. We don't want multiple threads to return and remove the top element from the stack. It is thread-safe operation.
d. E.g.

```
public class StackDemo {
        public static void main(String[] args) {
                Stack s=new Stack();
                s.push(10);
                s.push(20);
                s.push(30);
                s.push(40);

                System.out.println("Before pop: "+s);

                Object x=s.pop();

                System.out.println("After pop: "+s);
                System.out.println("Removed "+(Integer)x+" from the top");

        }
}
```

e. If pop method is invoked on empty stack, then we will get EmptyStackException.

3. Object peek():
   a. It returns the element at the top of the stack.
   b. It doesn't removes the element.
   c. E.g.

```
public class StackDemo {
        public static void main(String[] args) {
                Stack s=new Stack();
                s.push(10);
                s.push(20);
                s.push(30);
                s.push(40);

                System.out.println("Before peek: "+s);

                Object x=s.peek();

                System.out.println("After peek: "+s);

                System.out.println("Peek returned "+x);

        }

}
```

4. int search(Object o):
   a. It searches for the specified Object o in this Set, and returns its offset.
   b. Offset of element present on top is always 1.

```
public class StackDemo {
        public static void main(String[] args) {
                Stack s=new Stack();
                s.push(10);//Offset=4
                s.push(20);//Offset=3
                s.push(30);//Offset=2
                s.push(40); //Offset=1

                System.out.println("30 is located at "+s.search(30)+" offset");
        } }
```

   d. If element that we want to search is not present in this stack, then search() method will return -1.
   e. If there are duplicate elements in this stack, then it will return the offset of first occurrence of the element that we want to search.

5. boolean empty():
   a. It will return true if this stack is empty.

Cursors in Java:

- To iterate elements of collection, it recommended to not to use for loop
- To iterate any collection, java have provided Cursors.
- Cursors are used to iterate through any collection.
- It reduces chance of getting ConcurrentModificationException.
- There are three types of Cursors:
  1. Iterator
  2. ListIterator
  3. Enumeration
- All these cursors are interfaces in Java.

1. Iterator(I) 1.2v:

   a. It is used to iterate over any collection.
   b. It iterates only in forward direction.
   c. We cannot iterate collection in reverse direction.
   d. E.g.

```
public class ArrayListDemo {
        public static void main(String[] args) {
                Vector al = new Vector();
                al.add(10);
                al.add(20);
                al.add(30);
                al.add(40);
                al.add(50);
                al.add(60);

                Iterator itr = al.iterator();
                while (itr.hasNext()) {
                        System.out.println(itr.next());
                }
        }
```

Methods of Iterator:

1. boolean hasNext():
   - It checks whether this collection has elements in it.
   - If there are no more elements to iterate, then it will return false.
2. Object next():
   - It returns the next element from this collection and updates the pointer.
   - It will through NullPointerException if there are no more elements present in this collection.
3. void remove():
   - It removes the element returned by the iterator.

2. ListIterator(I): 1.2v:

   a. It is used to iterate through collection both in forward as well as in reverse direction.
   b. It is specifically designed for collection whose backend data structure is doubly Linked List.

Methods of ListIterator

1. boolean hasNext():
   - It will return true if there are elements present in this collection.
   - It will return false if there are no next elements in this collection.
2. Object next():
   - It will return the next element from this collection.
   - It will return NoSuchElementException if there are no next elements in this collection.
3. boolean hasPrevious():
   - It will return true if there are previous elements present in this collection
   - It will return false if there are no previous elements.
   - It can used to iterate collection in reverse order.
4. Object previous():
   - It is used to return previous element from this collection.
   - It will through NoSuchElementException if there are no previous elements in this collection.
5. int nextIndex():
   - It returns the index of element pointed by next() method.
   - It doesn't updates the pointer.
   - E.g.

```
public class ArrayListDemo {
	public static void main(String[] args) {
		Vector al = new Vector();
		al.add(10);
		al.add(20);
		al.add(30);
		al.add(40);
		al.add(50);
		al.add(60);

		ListIterator ltr = al.listIterator();
		while (ltr.hasNext()) {
			System.out.println("Index: " + ltr.nextIndex());
			System.out.println("Element: " + ltr.next());
		}
	}

}
```

6. int previousIndex():
   - It returns the index of element pointed by previous() method
7. void remove():
   - It removes the currently pointed element by next() method or by previous() method.
8. void Set(Object o):
   - It replaces the element currently pointed by next() or previous() method.
   - E.g.

```
public class ArrayListDemo {
	public static void main(String[] args) {
		Vector al = new Vector();
		al.add(10);
		al.add(20);
		al.add(30);
		al.add(40);
		al.add(50);
		al.add(60);
		ListIterator ltr = al.listIterator();
		System.out.println("Before Set operation: \n"+al+"\n");
		while (ltr.hasNext()) {
			if(ltr.next().equals(30)) {
				ltr.set(33);
			}
		}
		System.out.println("After Set operation: \n"+al);
	}
}
```

9. void add(Object o):
   - It adds the specified element at the next position to the element pointed by next() method.
   - It adds the specified element at the previous position to the element pointed by previous() method.

- E.g.
-

```
public class ArrayListDemo {
	public static void main(String[] args) {
		Vector al = new Vector();
		al.add(10);
		al.add(20);
		al.add(30);
		al.add(40);
		al.add(50);
		al.add(60);

		ListIterator ltr = al.listIterator();
		System.out.println("Before Add operation: \n"+al+"\n");
		while (ltr.hasNext()) {
			ltr.next();
		}

		while(ltr.hasPrevious()) {
			if(ltr.previous().equals(30)) {
				ltr.add(33);
			}
		}
		System.out.println("After Add operation: \n"+al);
	}

}
```

3. **Enumeration(I) 1.0v:**
    a. We can iterate only legacy collection classes using Enumeration.
    b. We cannot iterate elements in reverse order.
    c. We cannot remove elements using Enumeration.

Methods of Enumeration:

1. boolean hasMoreElements():
    a. It checks whether the given legacy collection has elements in it.
2. E nextElement():
    a. It will return the current element and update the pointer to next element of this legacy collection.

E.g. of Enumeration:

```
public class EnumearationDemo {
	public static void main(String[] args) {
		Vector v = new Vector

		v.add(10);
		v.add(20);
		v.add(30);
		v.add(40);
		v.add(50);

		Enumeration e=v.elements();
		while (e.hasMoreElements()) {
			System.out.println(e.nextElement());
		}
	}

}
```

Iterator	ListIterator	Enumeration
It is used to iterate any collection	It is specifically designed to iterate collection whose backend data structure is Doubly Linked List or child classes of List interface	It is used to iterate only legacy classes of collection
It can remove the element	It can remove the element	It cannot remove the element
It can iterate in only forward direction	It can iterate in forward as well as reverse direction	It can iterate only in forward direction
We cannot replace element	We can replace element using set() method	We cannot replace element
We cannot get index of element	We can get index of element using nextIndex() and previousIndex() methods	We cannot get index of element
Introduced in 1.2v of Java	Introduced in 1.2v of Java	Introduced in 1.0v of Java

Set (I):

- It is growable in nature
- It allows homogenous as well as heterogenous data elements.
- There are some Sets which doesn't allow heterogenous data elements. For example: TreeSet
- Insertion order is not guaranteed. But some Sets preserves insertion order. For example LinkedHashSet.
- Null insertion is possible in some of the Sets. TreeSet in Java 1.8 doesn't allow null.
-

Hashtable (Class):

- When we want to store data in key-value pair, then we can use Hashtable data structure.
- Java implemented Hashtable data structure using Hashtable class in version 1.0.
- Key and value can be homogenous or heterogenous.
- Values will be stored in Hashtable based on hash code of keys.
- It doesn't guarantee insertion order
- Key and value can't be empty
- Key and value both can't be null. If we add key or value as null, then we will get NullPointerException
- Keys can't be duplicate, whereas value can be duplicate
- All methods of Hashtable class are synchronized
- Hence Hashtable objects are thread safe.

Example of Hashtable:

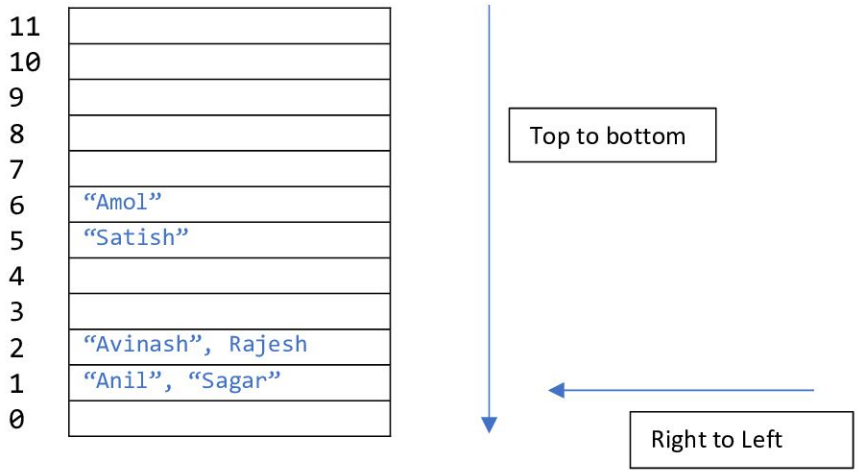
```
public class HashTableDemo {
        public static void main(String[] args) {
                Hashtable ht=new Hashtable(); //Default capacity is 11

                ht.put(1, "Anil");//bucket number=1
                ht.put(2, "Avinash"); //bucket number=2
                ht.put(5, "Satish"); //bucket number=5
                ht.put(17, "Amol"); //bucket number=17%11=6
                ht.put(23, "Sagar"); //bucket number=23%11=1
                ht.put(24, "Rajesh"); //bucket number=24%11=2

                System.out.println(ht);
        }

}
```

- In above program, default initial capacity of Hashtable ht is 11.
- So first element i.e. “Anil” will be added in 1st bucket.
- Second element will be added to second bucket
- Third element i.e. “Satish” will be added to 5th bucket
- Fourth element i.e. “Amol”, will be added to 6th bucket. Here there is no 17th number bucket, because capacity of ht is 11 buckets only. Hence the bucket number for “Amol” will be calculated using:

  bucket-number=hasCode(key)%Capacity

  E.g.

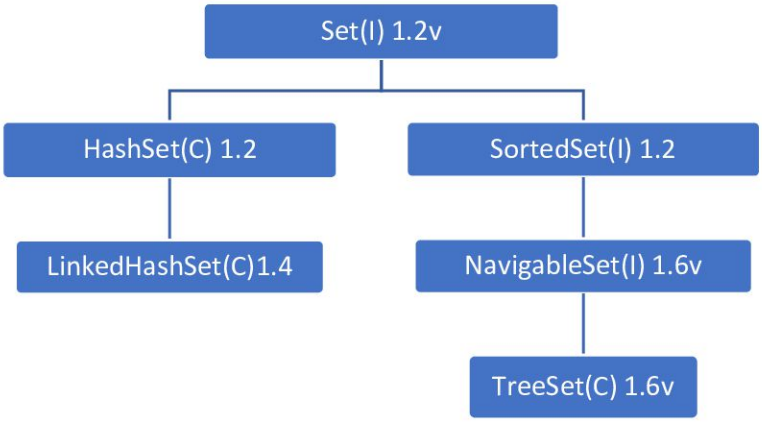
  bucket-number=17%11=6



- Output:
  [17=Amol, 5=Satish, 24=Rajesh, 2=Avinash, 23=Sagar, 1=Anil]
- If we increase the capacity of Hashtable then our output will change.
- Hashtable ht=new Hashtable(25);
- Then output will be:
  [24=Rajesh, 23=Sagar, 17=Amol, 5=Satish, 2=Avinash, 1=Anil]
- In Hashtable, reading is done from top to bottom. And if there are multiple elements in same bucket, then reading will be done in Right-to-Left manner.

- For integer type keys, hashCode is integer itself. For example, if key is 17 then its hashCode is also 17.
- Hashcode is calculated differently for different objects.
- For Integer type keys, hashcode is same as value.
- For String type keys, hascode will be calculated using following formula:
  `hashCode=s.charAt(0)*31^(size-1)+s.charAt(1)*31^(size-2)+s.charAt(3)*31^(size-3).....+s.charAt(n)*31^(size-n)`
- Primitive data elements doesn't have hashCode.

Hierarchy of Set interface:



Methods of Set interface:

1. Set interface doesn't have its own methods.
2. It uses all methods of Collection interface.

1) HashSet(C) 1.2v:

a) HashSet can store homogenous as well as heterogenous data elements.
b) Insertion order is not guaranteed.
c) Duplicate elements are not allowed. Even if we try to add duplicate elements, we won't get any error whereas add() method will return false.
d) Null insertion is possible only once.
e) It is growable in nature.
f) Its backend data structure is Hashtable.
g) Its default capacity is 16.
h) Performance for reading, insertion and deletion is constant.
i) HashSet is best for search operations.

```
public class HashSetDemo {
        public static void main(String[] args) {
                HashSet hs=new HashSet();//16
                hs.add(10);
                hs.add(17);
                hs.add(23);
                hs.add(46);
                hs.add(18);
                hs.add(10); //No error. Simply add method will return false
                System.out.println(hs);
        }
}
```

Sr.No	Hashtable	HashSet
1	It stores data in key-value pair	Single data elements are stored
2	Doesn't guarantees insertion order	Doesn't guarantees insertion order
3	Homogenous as well as heterogenous key or values are allowed.	Homegenous as well as heterogenous data elements are allowed
4	Key and value both cannot be null	Null is allowed only once
5	Duplicate keys are not allowed whereas values are allowed	Duplicate data elements are not allowed
6	All methods of Hashtable are synchronized.	None of the method of HashSet is synchronized
7	Hashtable is thread-safe	HashSet is not thread-safe
8	Introduced in Java v1.0	Introduced in Java v1.2
9	It is implementatioin class of Map interface	It is implementation class of Set interface
10	Its default capacity is 11	Its default capacity is 16

Constructors of HashSet:

1. HashSet():
   - It is used to create an empty HashSet of capacity 16 and load factor of 0.75 (75%).
   - Load Factor ? : It tells HashSet when to start grow. When HashSet capacity is 16, then it will start growing once 12 elements are added (16*(75/100)).
2. HashSet(int initialCapacity):
   - It is used to create an empty HashSet of specified initial capacity and load factor of 0.75.
3. HashSet(int initialCapacity, float loadFactor):
   - It is used to create an empty HashSet of specified initial capacity and specified load factor.
4. HashSet(Collection c):

- It is used to convert any Collection c into an HashSet.
- It will copy all elements from Collection c and store in HashSet object.

2) LinkedHashSet (C) 1.4v:

a) It is child class of HashSet
b) It doesn't have its own methods
c) It is growable
d) It can store homogenous as well as heterogenous data elements
e) It preserves insertion order
f) Its backed data structure is Double Linked List plus HashMap
g) Null insertion is possible only once.
h) Duplicates are not allowed.
i) Its initial capacity is 16 and load factor is 0.75

Constructors of LinkedHashSet:

1. LinkedHashSet():
   - It is used to construct an empty LinkedHashSet of initial capacity 16 and default load factor 0.75.
2. LinkedHashSet(int intialCapacity):
   - It is used to create an empty LinkedHashSet of specified initial capacity and default load factor of 0.75
3. LinkeHashSet(int initialCapacity, float loadFactor):
   - It is used to create empty LinkedHashSet with specified initial capacity and specified load factor.
4. LinkedHashSet(Collection c):
   - It is used to convert any collection c into a LinkedHashSet object.
   - In-short it copies all elements from Collection c into an LinkedHashSet objects.

SortedSet(I) 1.2v:

- When we want to store unique elements in some sorting order, then we can use SortedSet.
- It is growable in nature.
- Duplicates are not allowed.
- It will sort elements in Default Natural Sorting Order (DNSO).
- Heterogenous elements are NOT allowed. If programmer tries to insert heterogenous element, then he/she may get ClassCastException
- Insertion order is not preserved.

Methods of SortedSet:

1. SortedSet subset(Object fromElement, Object toElement):
   a. It will return all elements which are greater-than-or-equal-to fromElement and less-than toElement.
   b. E.g.

```
public class TreeSetDemo {

	public static void main(String[] args) {
		TreeSet t=new TreeSet();
		t.add(10);
		t.add(50);
		t.add(30);
		t.add(20);
		t.add(40);
		t.add(70);

		System.out.println(t.subSet(10, 35));//Output=>[10,20,30]
	}
}
```

2. SortedSet headSet(Object toElement):
    a. It will return Sorted set of elements which are less than toElement.
    b. E.g.

```
public class TreeSetDemo {

	public static void main(String[] args) {
		TreeSet t=new TreeSet();
		t.add(10);
		t.add(50);
		t.add(30);
		t.add(20);
		t.add(40);
		t.add(70);

		System.out.println(t.headSet(30));//Output=>[10,20]
	}
}
```

3. SortedSet tailSet(Object fromElement):
    a. It will return Sorted set of elements which are greater-than-and-equals-to fromElement.
    b. E.g

```
public class TreeSetDemo {

	public static void main(String[] args) {
		TreeSet t=new TreeSet();
		t.add(10);
		t.add(50);
		t.add(30);
		t.add(20);
		t.add(40);
		t.add(70);

		System.out.println(t.tailSet(22));//Output=>[30,40,50,70]
	}
}
```

4. Object first():
    a. It will return the first element from the Sorted set. In short it will return the smallest element from the set.
    b. E.g.

```
public class TreeSetDemo {

        public static void main(String[] args) {
                TreeSet t=new TreeSet();
                t.add(10);
                t.add(50);
                t.add(30);
                t.add(20);
                t.add(40);
                t.add(70);

                System.out.println(t.first());//Output=>[10]
        }
}
```

5. Object last():
    a. It will return the last element from this Sorted Set.
    b. In short it will return biggest element.

Comparable(I):

- If we want to sort two primitive data types, we can compare them by using operators like: less-than (<), greatert-than (>) and equal-to(==).
- But what if we want to compare and sort two Student class objects in ascending or descending order?
- We cannot compare Student objects using less-than or greater-than operators. Objects can be compared using equals() method of Object class only. But equals method compares hashcode of the objects.
- Hence Java provided Comparable interface to provide an ability of comparison to the class object.
- Objects of class who implements Comparable interface can be compared and sort in ascending or descending order.
- Comparable class have compareTo() method. By implementing this method we can arrange any class object in ascending or descending order based on some property of that object.
- Consider below example

```
public class Student {
        private String name;
        private float marks;
        public Student(String name, float marks) {
                this.name=name;
                this.marks=marks;
        }
}
```

- If I want to sort Student class objects in ascending order using Collections.sort() method, then I will get RE:ClassCastException
- Consider below program

```
public class ComparableDemo {
	public static void main(String[] args) {
		Student s1=new Student("Akash",75f);
		Student s2=new Student("Satish",86f);
		Student s3=new Student("Rajesh",93f);
		Student s4=new Student("Bhushan",68f);

		ArrayList<Student> al=new ArrayList<Student>();
		al.add(s1);
		al.add(s2);
		al.add(s3);
		al.add(s4);


		Collections.sort(al);
		System.out.println("After Sorting by marks: \n");
		for(Student s:al) {
			System.out.println(s.name);
		}

	}

}
```

- In above program, we will get an error: ClassCastException on line Collections.sort().
- But if we implement Comparable interface in Student class, then error will be gone and it will sort all students by their marks.
- E.g.

```
public class Student implements Comparable{
	public String name;
	public float marks;
	public Student(String name, float marks) {
		this.name=name;
		this.marks=marks;
	}
	@Override
	public int compareTo(Object o) {
		Student x=(Student)o;
		if(this.marks<x.marks)
			return -1;
		else if (this.marks>x.marks)
			return 1;
		else
			return 0;
	}
}
```

- Now if we run ComparableDemo class, then all Student class objects s1,s2,s3 na s4 will be sorted in ascending order.

compareTo(Object o):

- It belongs to Comparable interface.
- it returns int.
- E.g. x.compareTo(y).
- iff x is less than y, then it returns -ve int value
- iff x is greater than y, then it return +ve int value
- iff x is equal to y then it returns 0.
- If we implement compareTo() method as per above rules, then we get objects sorted in ascending order.
- If we want to sort objects in descending order, then we have to reverse the implementation of compareTo() method in Student class as shown below.

```
public class Student implements Comparable{
        public String name;
        public float marks;
        public Student(String name, float marks) {
                this.name=name;
                this.marks=marks;
        }
        @Override
        public int compareTo(Object o) {
                Student x=(Student)o;
                if(this.marks<x.marks)
                        return 1;
                else if (this.marks>x.marks)
                        return -1;
                else
                        return 0;
        }
}
```

- Here now, all Student objects will be compared on basis of marks.
- But what if we want to sort Student objects on basis of their names?
- Then we have to remove this implementation of compareTo() method from Student class, and add another implementation which compares Student's name.
- Because we earlier have given an ability to Student object to compare by marks. We need to change this ability now.
- But what if we want to compare Student objects sometimes on basis of name and sometimes on basis of marks?
- Obviously, we cannot achieve this because we cannot have two implementations of compareTo() method in Student class.
- To achieve such scenario, Java have provide Comparator interface, which externally compares two objects.

Comparator:

1. It is an interface is Java, which helps to compare two homogenous objects externally on basis of some feature of those objects.
2. Consider following Student class.

```
public class Student{
	public float marks;
	public String name;
	public float height;
	public float feesBalance;

	public Student(float marks, String name, float height, float feesBalance) {
		this.marks = marks;
		this.name = name;
		this.height = height;
		this.feesBalance=feesBalance;
	}
}
```

3. E.g.

```
public class TreeSetDemo {

	public static void main(String[] args) {

		Student ramesh=new Student(76.77f, "Ramesh", 4.8f, 10000f);
		Student suresh=new Student(46.77f, "Suresh", 4.4f, 5000f);
		Student kalia=new Student(96.77f, "Kalia", 4.2f, 10000f);
		Student gabbar=new Student(86.77f, "Gabbar", 3.8f, 7000f);
		Student samba=new Student(53.77f, "Samba", 4.1f, 3000f);
		ArrayList<Student> al=new ArrayList<Student>();
		al.add(ramesh);
		al.add(suresh);
		al.add(kalia);
		al.add(gabbar);
		al.add(samba);

		Collections.sort(al);
		System.out.println("After Sorting by marks: \n");
		for(Student s:al) {
			System.out.println(s.name);
		}
	}
}
```

4. In above program, we will get RE:ClassCastException.
5. Because, Student class is not comparable.
6. Hence we have created Comparator to compare two students objects on basis of Marks.
7. Observe below Comparator

```
public class CompareOnMarks implements Comparator{

	@Override
	public int compare(Object o1, Object o2) {
		Student x=(Student)o1;
		Student y=(Student)o2;
		return x.marks<y.marks ? -1:(x.marks==y.marks) ? 0:1;
	}

}
```

8. Now if we provide this Comparator to the Collections.sort() method, then we will get list of students sorted in ascending order.

```
public class TreeSetDemo {

        public static void main(String[] args) {
                CompareOnMarks sortByMarks=new CompareOnMarks();

                Student ramesh=new Student(76.77f, "Ramesh", 4.8f, 10000f);
                Student suresh=new Student(46.77f, "Suresh", 4.4f, 5000f);
                Student kalia=new Student(96.77f, "Kalia", 4.2f, 10000f);
                Student gabbar=new Student(86.77f, "Gabbar", 3.8f, 7000f);
                Student samba=new Student(53.77f, "Samba", 4.1f, 3000f);
                ArrayList<Student> al=new ArrayList<Student>();
                al.add(ramesh);
                al.add(suresh);
                al.add(kalia);
                al.add(gabbar);
                al.add(samba);


                Collections.sort(al, sortByMarks);
                System.out.println("After Sorting by marks: \n");
                for(Student s:al) {
                        System.out.println(s.name);
                }

        }
}
```

TreeSet(C):

- It is used to store homogenous and comparable objects only
- Heterogenous elements are not allowed
- Duplicates are not allowed
- Insertion order is not preserved.
- Its backend data structure is Red-Black Balanced Tree
- Null insertion is not possible since Java1.8. If we try to insert null in TreeSet then we will get RE:NullPointerException
- It sorts elements in DNSO or in a order provided by Comparator.

Constructors of TreeSet:

1. TreeSet():
    a. It constructs an empty TreeSet which can have only homogenous and comparable elements in it.
    b. It we try to insert elements which doesn't implement Comparable, then we will get, RE:ClassCastException.
2. TreeSet(Collection c):
    a. It is used to convert any Collection c into a TreeSet.
    b. In short it copies all unique elements into a TreeSet and sorts them in DNSO.
3. TreeSet(Comparator c):
    a. It is used to create an empty TreeSet whose all elements will be sorted in the order provided by Comparator c.

   b. It will allow only homogenous elements, but not required to have Comparable implemented.
   c. Because we are providing external comparator to compare two object to insert.
4. TreeSet(SortedSet s):
   a. It is used to create an empty TreeSet who has elements from SortedSet.
   b. There is a child class of Sorted set, ConcurrentSkipListSet. If we want to convert ConcurrentSkipListSet into a TreeSet, then we can use this constructor.
   c. E.g.

```
public class TreeSetDemo {

        public static void main(String[] args) {
                SortedSet ts=new ConcurrentSkipListSet<>();

                ts.add(10);
                ts.add(30);
                ts.add(15);
                ts.add(25);
                ts.add(5);
                TreeSet t=new TreeSet(ts);

                System.out.println(ts); //[5,10,15,25,30]

        }
}
```

5. TreeSet(NavigableMap m):
   a. It is used to convert NavigableMap into a TreeSet.

Methods of TreeSet:

1. Object lower(Object x):
   a. It will return higher element from the set of elements which are lesser than the specified the Object x.
   b. E.g.
   c. If we invoke lower() method on above mentioned TreeSet then we will get:
   d. ts.lower(27): It will return 25. Because 25 is higher from the set [5,10,15,25]. This set contains elements which are lesser than 27.
   e. If there is no element matching, then it will return null.
2. Object higher(Object x):
   a. It will return lower element from the set of elements which are higher than the specified element x.
   b. E.g. ts.higher(13): It will return 15.
3. Object floor(Object x):
   a. It will return highest element from a set of elements which are less than or equal to specified element x.
   b. E.g. ts.floor(15): It will return 15, because first it will create a set [5,10,15]. Then it will return highest element from that set, that is 15.

c. E.g. ts.floor(17): It will return 15, because first it will create a set [5,10,15]. Then it will return highest element from that set, that is 15.

4. Object ceiling(Object x):
   a. It will return lowest element from a set of elements which are greater than or equals to specified element x.
   b. E.g. ts.ceiling(17): It will return 25. First it will create a set [25,30]. Then it will return lowest element from this set, that is 25.
   c. E.g. ts.ceiling(15): It will return 15. First it will create a set [15,25,30]. Then it will return the lowest element that is 15.
5. NavigableSet descendingSet():
   a. It will return reverse ordered Navigable set from a given set.
   b. If original set is in ascending order, then this method will return set in descending order and vice a versa.
   c. For Example

```
public class TreeSetDemo {

        public static void main(String[] args) {
                TreeSet ts=new TreeSet();

                ts.add(10);
                ts.add(30);
                ts.add(15);
                ts.add(25);
                ts.add(5);
                System.out.println("Before descendingSet() method: "+ts);
                System.out.println("After descendingSet() method: "+ts.descendingSet());
        }
}
```

   d. This method originally belongs to NavigableSet interface.
6. NavigableSet subset(Object fromElement, boolean fromInclusive, Object toElement, boolean toInclusive):
   a. It is same as subset() method of SortedSet interface. But in NavigableSet, this method is highly customizable.
   b. It is overloaded here in NavigableSet
   c. We can configure whether we want to include fromElement and toElement in the resulting subset.
   d. E.g. Consider there is a TreeSet ts[5,10,15,25,27,32,48]
   e. Observe below cases
      i. Case 1:
         1. subSet(5,false, 32, true): O/P => [10,15,25,27,32]
      ii. Case 2:
         1. subSet(5,true,32,false): O/P => [5,10,15,25,27]
      iii. Case 3:
         1. subset(5,true, 32, true): O/P => [5,10,15,25,27,32]
      iv. Case 4:
         1. subSet(5, false,32,false): O/P => [10,15,25,27]
7. NavigableSet headSet(Object toElement, boolean inclusion):
   a. It is same as headSet() method of SortedSet interface. But in NavigableSet, it is more customizable.

b. Here we can set whether we want to include toelement or not.
c. For Example: Consider a TreeSet with elements ts => [5,10,15,25,27,32,48]
d. Observe below cases:
    i. Case 1:
        1. ts.headSet(25,true): O/P => [5,10,15,25]
    ii. Case 2:
        1. ts.headSet(25, false): O/P => [5,10,15]

8. NavigableSet tailSet(Object fromElement, boolean inclusion):
    a. It is same as tailSet() method of SortedSet where as it is more customizable in NavigableSet
    b. Here we can decide whether we want to include fromElement or not.
    c. For Example: Consider a TreeSet with elements ts => [5,10,15,25,27,32,48]
    d. Observe below cases:
        i. Case 1:
            1. ts.tailSet(25,true): O/P => [25,27,32,48]
        ii. Case 2:
            1. ts.tailSet(25,false): O/P => [27,32,48]
    e. This method originally belongs to NavigableSet interface.
9. Object pollFirst():
    a. It will return and remove the first element from this Set.
    b. E.g

```
public class TreeSetDemo {

        public static void main(String[] args) {
                TreeSet ts=new TreeSet();

                ts.add(10);
                ts.add(15);
                ts.add(32);
                ts.add(15);
                ts.add(25);
                ts.add(5);
                ts.add(27);
                ts.add(48);

                System.out.println(ts.pollFirst());
                System.out.println("After pollFirst() method: "+ts);
        }
}
```

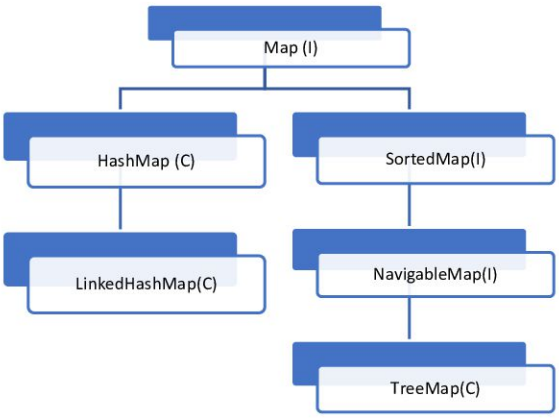
10. Object pollLast():
    a. It returns and removed the last element from this Set.
    b. E.g.

```
public class TreeSetDemo {

	public static void main(String[] args) {
		TreeSet ts=new TreeSet();

		ts.add(10);
		ts.add(15);
		ts.add(32);
		ts.add(15);
		ts.add(25);
		ts.add(5);
		ts.add(27);
		ts.add(48);
		System.out.println("Before pollLast() Method: "+ts);
		System.out.println("Last element: "+ts.pollLast());
		System.out.println("After pollLast() method: "+ts);
	}
}

/* OUTPUT

Before pollLast() Method: [5, 10, 15, 25, 27, 32, 48]
Last element: 48
After pollLast() method: [5, 10, 15, 25, 27, 32]

*/
```

Map:

- When we want to store data in key-value pair, the we can consider using Map.
- Map cannot have duplicate keys, whereas values can be duplicate.
- Map can have only one null key, whereas multiple null values are allowed.
- Insertion order is not guaranteed in Map.
- Elements are stored on basis of HashCode of the keys.
- It is growable in nature
- Both keys and values cane be Homogenous as well as Heterogenous. Except TreeMap where keys must be Homogenous and Comparable.
- It has ready-made method support available.
- Different Data structures are implemented in Map.
- Each pair of key-value is called as Entry. To represent Entry, Map has inner interface with same name i.e Entry

Hierarchy of Map:



Methods of Map:

1. int size():
   - It returns the number of entries present in this map.
2. boolean isEmpty():
   - It returns true iff this map is empty and having no entries in it
3. Object put(Object key, Object value):
   - It inserts given key-value pair in this map.
   - If the key is already present, then it will update the old value with new value and will return the old value.
4. Object get(Object key):
   - It returns the value associated with the given key.
   - It will return 'null' if the key is NOT present in this map.
5. boolean containsKey(Object key):
   - It returns true iff the specified key is present in this Map.
6. boolean containsValue(Object value):
   - It returns true iff the specified value is present in this Map.
   - It will only check the first occurrence of the value.
7. Object remove(Object key):
   - It will remove the key-value mapping associated with specified key.
   - It will return the previous value associated with the key
8. Set keySet():
   - It returns Set of keys present in this Map.
   - If this Map is empty, then Set will have null.
9. Set entrySet():
   - It returns Set of Entries present in this Map
   - Where Entry is a key-value pair
10. Collection values():
    - It reurns the Collection of values present in this Map
11. void putAll(Map m):
    - It inserts all entries from Map 'm' into this Map
12. void clear():

- It will remove all Entries from this Map.

1**. HashMap:**

- It allows homogenous as well as heterogenous keys and values
- Duplicate keys are not allowed whereas duplicate values are allowed
- Insertion order is not guaranteed. Elements are inserted as per the order of the hashcode of the keys.
- It allows null key only once whereas null value is allowed multiple times.
- Its backend data structure is Hashtable
- Its default capacity is 16.

Constructors of HashMap:

1. public HashMap():
    a. It is used to create an empty HashMap with initial capacity 16 and default load factor as 0.75
2. public HashMap(int initialCapacity):
    a. It is used to create an empty HashMap with specified initial capacity and default load factor of 0.75
3. public HashMap(int initialCapacity, float loadFactor):
    a. It is used to create an empty HashMap of specified initial capacity and specified load factor.
4. public HashMap(Map m):
    a. It is used to create an HashMap which will contain all elements from specified Map m.
    b. In short it can be used to convert any Map into HashMap.

3. LinkedHashMap:

- It allows homogenous as well as heterogenous keys and values
- Insertion order will be preserved.
- Duplicate keys are not allowed whereas duplicate values are allowed
- null key is allowed only once whereas null values are allowed more than once.
- Backend data structure is doubly-LinkedList plus Hashtable.
- It is slow for reading operations whereas fast for insertion and deletion operations

Constructors of LinkedHashMap:

1. LinkedHashMap():
    a. It is used to create an empty LinkedHashMap which will preserve order of eleemnts based on their keys.
    b. Default initial capacity will be 16
    c. Default load factor will be of 0.75
2. LinkedHashMap(int initialCapacity):
    a. It is used to create an empty LinkedHashMap of specified intial capacity
    b. Default load factor will be of 0.75
3. LinkedHashMap(int initialCapacity, float loadFactor):

a. It is used to create an empty LinkedHashMap of specified intial capacity and specified load factor

4. LinkedHashMap(Map m):
   a. It is used to create an LinkedHashMap which will contain elements from Map m.
   b. It will not create an empty LinkedHashMap as long as Map m is empty.

```
public class MapDemo {
        public static void main(String[] args) {

                HashMap lm = new HashMap<>();
                lm.put(2, "Amar");
                lm.put(1, "Akbar");
                lm.put(4, "Vijay");
                lm.put(3, "Dinanath");
                LinkedHashMap lhm = new LinkedHashMap(lm);
                System.out.println(lhm);

        }
}
```

4.TreeMap:

- Data is stored in the form of key-value pair
- Keys should be homogenous and Comparable
- Values can be heterogenous
- Keys cannot be null
- Values can be null
- It sorts all Entries based on Comparable order of keys.
- It doesn't allow Heterogenous keys. If we try to insert heterogenous keys then we will get ClassCastException.
- It doesn't preserves insertion order. Whereas it will Sort all Entries based on the order mentioned in compareTo() method.

Constructors of TreeMap:

1. TreeMap():

- It will create an empty TreeMap
- This TreeMap will sort all keys based on the order mentioned in their compareTo() method.
- If we try to put heterogenous keys then we will get ClassCastException
- If we try to put Homogenous keys but not Comparable keys then we will get ClassCastException
- If we try to put null key then we will get NullPointerException

2. TreeMap(Comparator c):

- It will sort all keys on basis of Comparator specified as an argument to this constructor.

```
public class TreeMapDemo {
	public static void main(String[] args) {
		Student s1 = new Student(1, "Vijay", "Chauhan", 78, "OBC", 6.6f);
		Student s2 = new Student(1, "Ganesh", "Gaitonde", 48, "OBC", 4.5f);
		Student s3 = new Student(1, "Sartaj", "Singh", 68, "OBC", 6.2f);


		TreeMap tm = new TreeMap<>(new MarksComparator());
		tm.put(s1, "Avinash");//Integer
		tm.put(s2, "Gaitonde");
		tm.put(s3, null);

		System.out.println(tm);
	}
}
```

```
public class Student{

	public int rollNumber;
	public String fname;
	public String lname;
	public int marks;
	float height;
	public String category;
	public Student(int rollNumber, String fname, String lname, int marks,String category,float height)
{
		super();
		this.rollNumber = rollNumber;
		this.fname = fname;
		this.lname = lname;
		this.category = category;
		this.marks = marks;
		this.height = height;
	}

}
```

```
public class MarksComparator implements Comparator{

	@Override
	public int compare(Object o1, Object o2) {
		Student s = (Student)o1;
		Student t = (Student)o2;


		if(s.marks>t.marks) {
			return 1;
		}else if(s.marks<t.marks) {
			return -1;
		}else {
			return 0;
		}
	}
```

3. TreeMap(Map m):
   - It is used to create a TreeMap from any Map having Homogenous and Comparable keys.

Sr.No	HashMap	Hashtable
1	It stores data in key-value pair	It stores data in key-value pair
2	It allows homogenous as well as heterogenous keys and values	It allows homogenous as well as heterogenous keys and values
3	It doesn't guarantees insertion order	It doesn't guarantees insertion order
4	Duplicate keys are not allowed whereas duplicate values are allowed	Duplicate keys are not allowed whereas duplicate values are allowed
5	Null key is allowed only once	Null key is NOT allowed
6	Null value is allowed multiple times	Null value is NOT allowed
7	It is not thread-safe	It is thread-safe
8	None of the methods of HashMap is synchronized	All methods of Hashtable are Synchronized