Hashtable class

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

V **getOrDefault**(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.

```
Hashtable<Integer,String> map=new Hashtable<Integer,String>();
 map.put(100,"Amit");
 map.put(102,"Ravi");
 map.put(101,"Vijay");
 map.put(103,"Rahul");

 System.out.println(map.getOrDefault(101, "Not Found"));
 System.out.println(map.getOrDefault(105, "Not Found"));
```

Output:

```
Vijay
Not Found
```

V **putIfAbsent**(K key, V value)	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

```
Hashtable<Integer,String> map=new Hashtable<Integer,String>();
 map.put(100,"Amit");
 map.put(102,"Ravi");
 map.put(101,"Vijay");
 map.put(103,"Rahul");
```

```
    System.out.println("Initial Map: "+map);

    map.putIfAbsent(104,"Gaurav");
    System.out.println("Updated Map: "+map);

    map.putIfAbsent(101,"Vijay");
    System.out.println("Updated Map: "+map);
```

Output:

```
Initial Map: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}
```

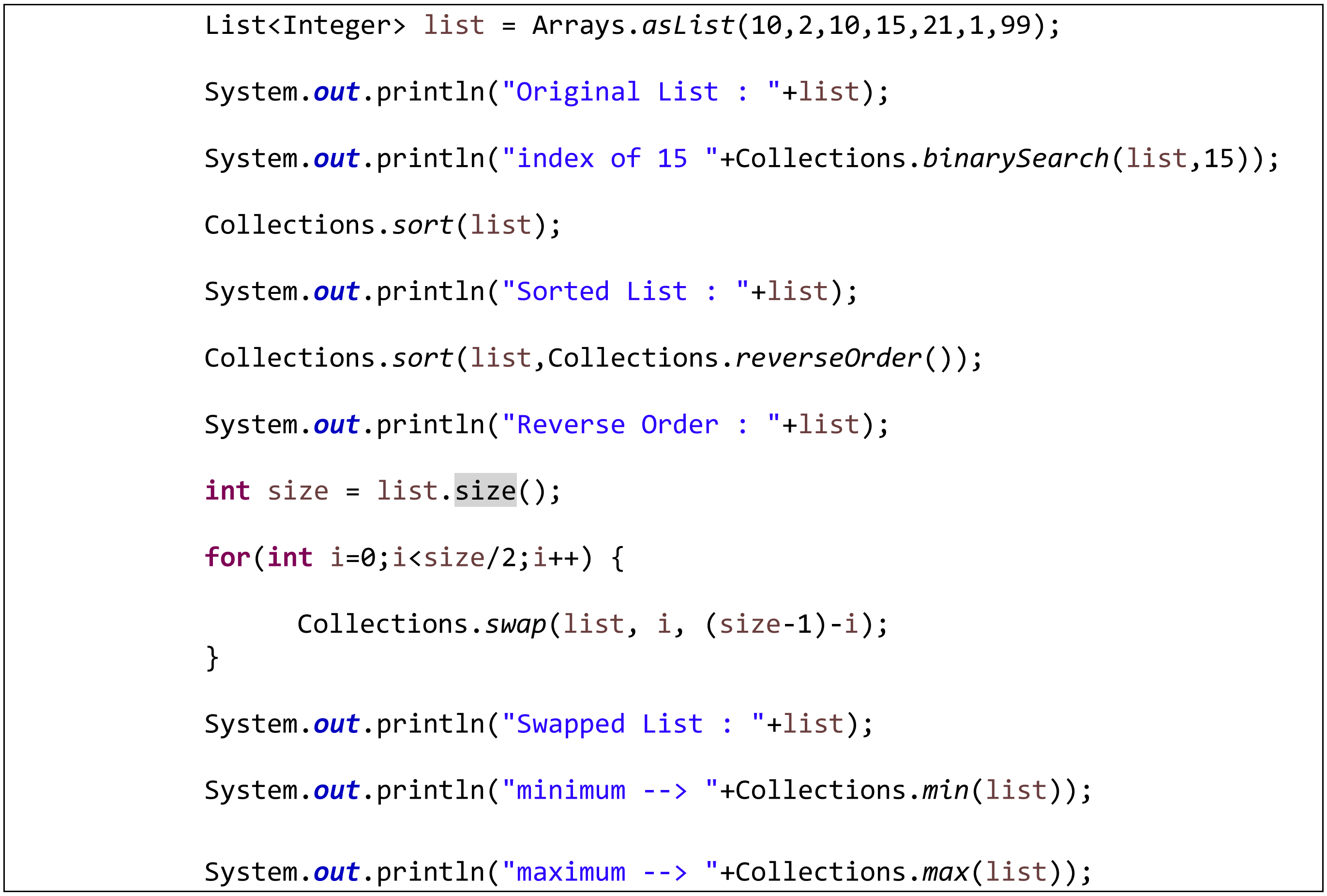
HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.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

Collections class **java.util.Collections**

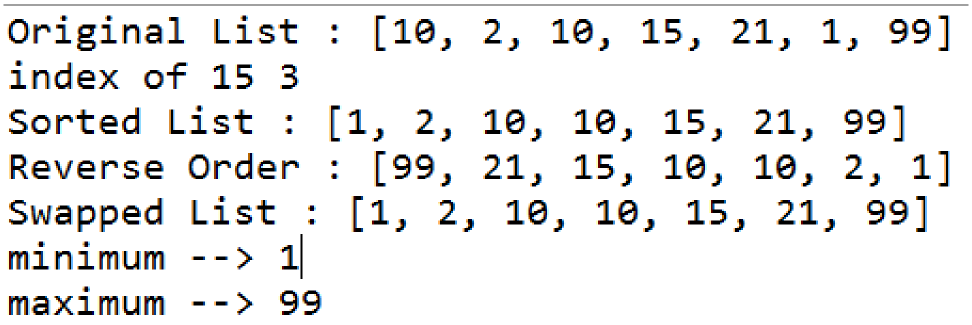
collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

The important points about Java Collections class are:

- Java Collection class supports the **polymorphic algorithms** that operate on collections.
- Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.



```java
List<Integer> list = Arrays.asList(10,2,10,15,21,1,99);

System.out.println("Original List : "+list);

System.out.println("index of 15 "+Collections.binarySearch(list,15));

Collections.sort(list);

System.out.println("Sorted List : "+list);

Collections.sort(list,Collections.reverseOrder());

System.out.println("Reverse Order : "+list);

int size = list.size();

for(int i=0;i<size/2;i++) {

      Collections.swap(list, i, (size-1)-i);
}

System.out.println("Swapped List : "+list);

System.out.println("minimum --> "+Collections.min(list));


System.out.println("maximum --> "+Collections.max(list));
```



```
Original List : [10, 2, 10, 15, 21, 1, 99]
index of 15 3
Sorted List : [1, 2, 10, 10, 15, 21, 99]
Reverse Order : [99, 21, 15, 10, 10, 2, 1]
Swapped List : [1, 2, 10, 10, 15, 21, 99]
minimum --> 1
maximum --> 99
```

Comparable interface

Comparable interface is used to order the objects of the user-defined class

It contains only one method named compareTo(Object)

It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

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

> *Note: String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.*

```java
public class Student implements Comparable<Student>{

      private int studentId;
      private String studentName;
      private String studentAddress;

      public Student(int studentId, String studentName, String studentAddress) {

            this.studentId = studentId;
            this.studentName = studentName;
            this.studentAddress = studentAddress;
      }

      @Override
      public String toString() {
            return studentId+"-"+studentName+"-"+studentAddress;
      }

      @Override
      public int compareTo(Student o) {

            if(this.studentId < o.studentId)
                  return -1;
            else if(this.studentId > o.studentId)
                  return 1;
            else
                  return 0;
      }
}
```

```
public class StudentList {

	public static void main(String[] args) {

		List<Student> list = new ArrayList<>();

		list.add(new Student(15,"Akash","Chennai"));
		list.add(new Student(20,"Pushpak","Patna"));
		list.add(new Student(2,"Anamika","Raipur"));
		list.add(new Student(3,"Pawan","Indore"));

		System.out.println("Original List : "+list);

		Collections.sort(list);

		System.out.println("Sorted List : "+list);

	}
}
```

```
Original List : [15-Akash-Chennai, 20-Pushpak-Patna, 2-Anamika-Raipur, 3-Pawan-Indore]
Sorted List : [2-Anamika-Raipur, 3-Pawan-Indore, 15-Akash-Chennai, 20-Pushpak-Patna]
```

Comparator interface

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

This interface is found in java.util package and contains 2 methods **compare**(Object obj1,Object obj2) and **equals**(Object element).

It **provides multiple sorting sequences**, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

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

Method of Collections class for sorting List elements

public void **sort(List** list**, Comparator** c**):** is used to sort the elements of List by the given Comparator

```java
public class Student {

      public int studentId;
      public String studentName;
      public String studentAddress;

      public Student(int studentId, String studentName, String studentAddress) {

            this.studentId = studentId;
            this.studentName = studentName;
            this.studentAddress = studentAddress;
      }

      @Override
      public String toString() {
            return studentId+"-"+studentName+"-"+studentAddress;
      }
}
```

```java
public class StudentComparator {

      public static void main(String[] args) {

            List<Student> list = new ArrayList<>();

            list.add(new Student(15,"Akash","Chennai"));
            list.add(new Student(20,"Pushpak","Patna"));
            list.add(new Student(2,"Anamika","Raipur"));
            list.add(new Student(3,"Pawan","Indore"));

            System.out.println("Original List : "+list);

            Comparator<Student> compareByName = (s1,s2) -> {
                  return s1.studentName.compareTo(s2.studentName);
            };

            Collections.sort(list,compareByName);

            System.out.println("Sorted by Name : "+list);

            Comparator<Student> compareById = (s1,s2) -> {
                  if(s1.studentId < s2.studentId)                return -1;
                  else if(s1.studentId > s2.studentId)           return 1;
                  else                    return 0;
            };

            Collections.sort(list,compareById);

            System.out.println("Sorted by Id : "+list);
      }
}
```

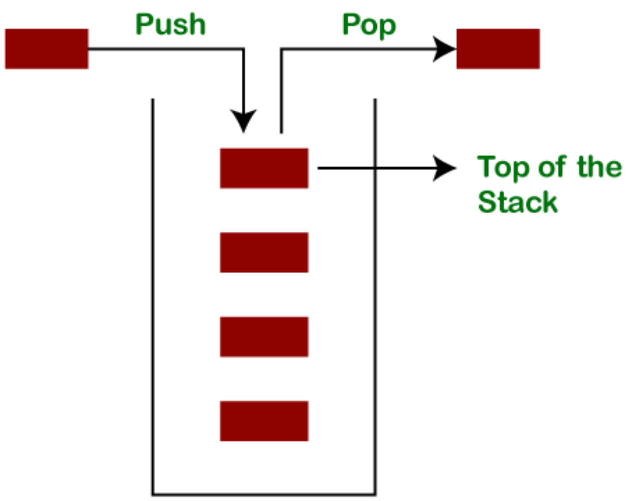
```
Original List : [15-Akash-Chennai, 20-Pushpak-Patna, 2-Anamika-Raipur, 3-Pawan-Indore]
Sorted by Name : [15-Akash-Chennai, 2-Anamika-Raipur, 3-Pawan-Indore, 20-Pushpak-Patna]
Sorted by Id : [2-Anamika-Raipur, 3-Pawan-Indore, 15-Akash-Chennai, 20-Pushpak-Patna]
```

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order.

ArrayList	Vector
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.

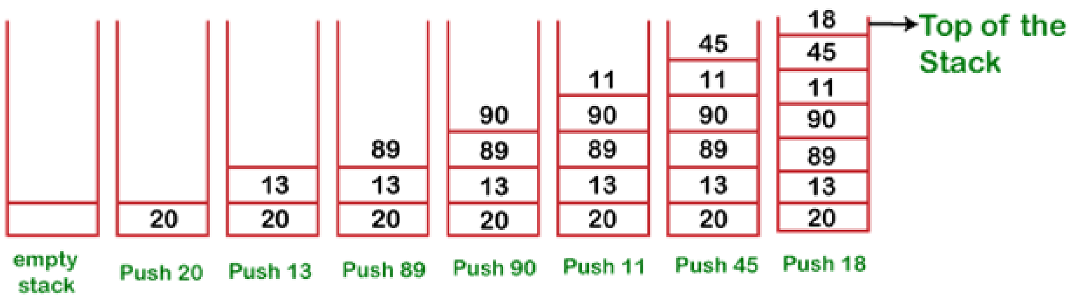


Stack

The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO)

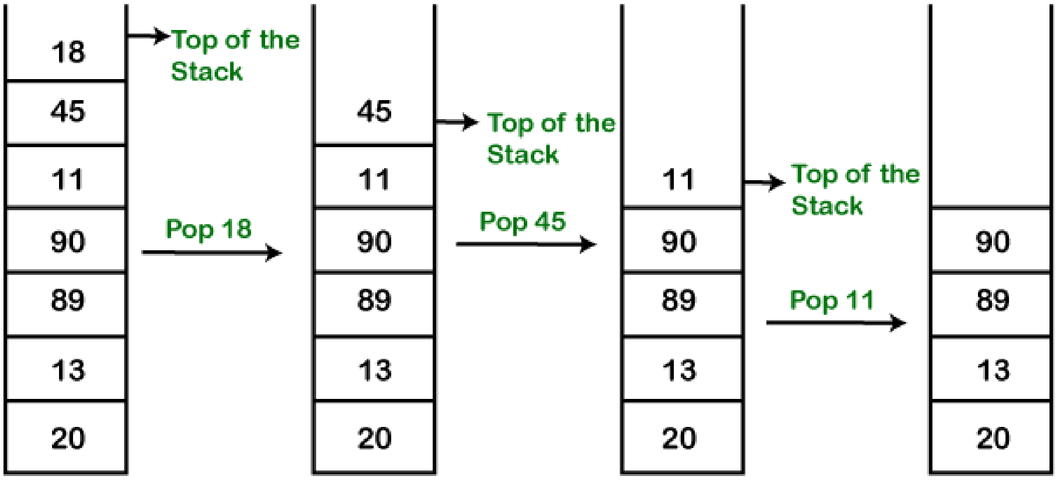
The **push** operation inserts an element into the stack and **pop** operation removes an element from the top of the stack

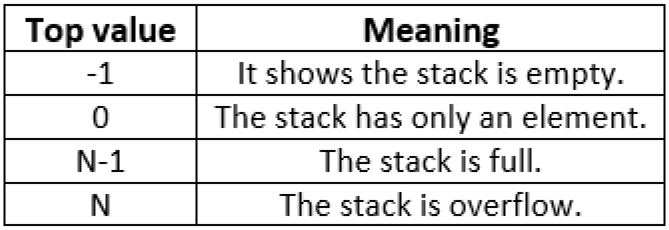
Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



Push operation

Let's remove (pop) 18, 45, and 11 from the stack.





Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.

Empty Stack: If the stack has no element is known as an **empty stack**. When the stack is empty the value of the top variable is -1
When we push an element into the stack the top is **increased by 1**.

- Push 12, top=0
- Push 6, top=1
- Push 9, top=2

When we pop an element from the stack the value of top is **decreased by 1**.

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function. It throws **EmptyStackException** if the stack is empty
peek()	E	The method looks at the top element of the stack **without removing it**. It also throws **EmptyStackException** if the stack is empty
search(Object o)	int	The method searches the specified object and returns the position of the object. It uses **equals()** method to search an object in the stack.

Traversing using ListIterator for stack:-

Using listIterator(int i) Method

This method returns a list iterator over the elements in the mentioned list (in sequence), starting at the specified position in the list. **It iterates the stack from top to bottom**

It throws **IndexOutOfBoundsException** if the index is out of range.

```
Stack<Integer> stack = new Stack<>();

stack.push(100);

stack.push(200);

stack.push(400);

stack.push(800);

System.out.println("Printing using for loop");

for(Integer i:stack){

    System.out.print(i+" ");

}

System.out.println();

System.out.println("Printing using List Iterator");

ListIterator<Integer> list = stack.listIterator(stack.size());

while(list.hasPrevious()){

    System.out.print(list.previous()+" ");

}
```

OUTPUT:-

```
Printing using for loop

100 200 400 800

Printing using List Iterator

800 400 200 100
```