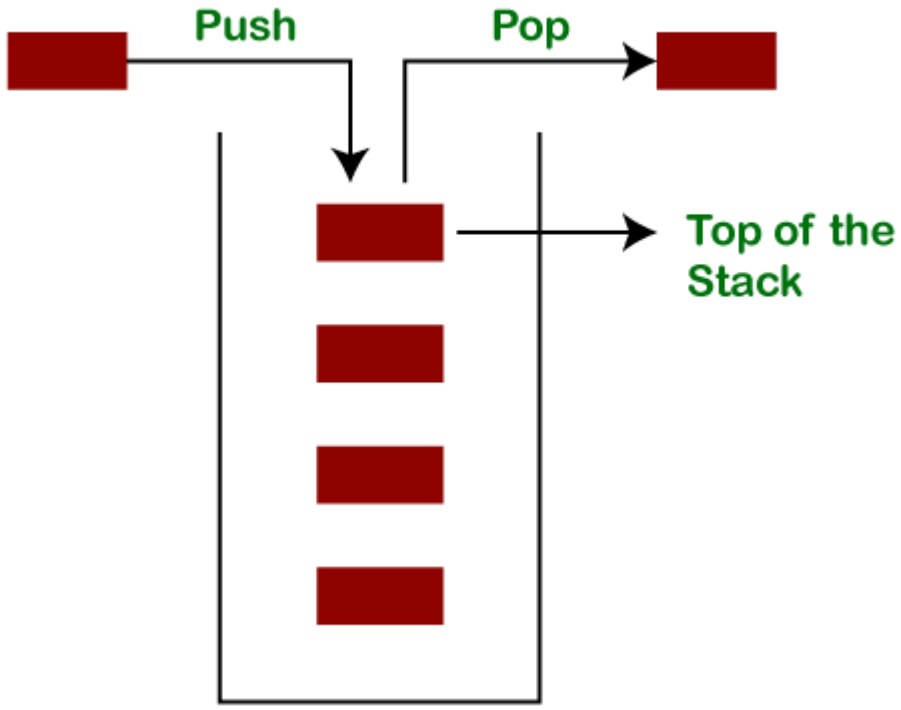
Java Stack

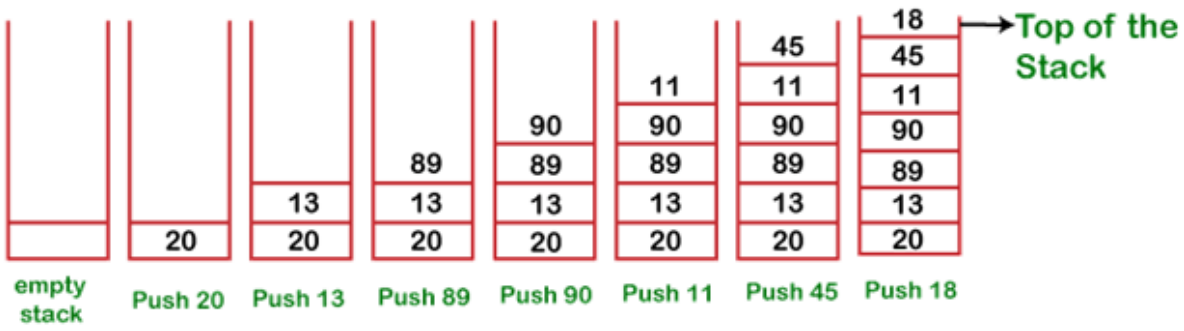
The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

In this section, we will discuss the **Java Stack class**, its **methods,** and **implement** the stack data structure in a Java program. But before moving to the Java Stack class have a quick view of how the stack works.

The stack data structure has the two most important operations that are **push** and **pop**. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.

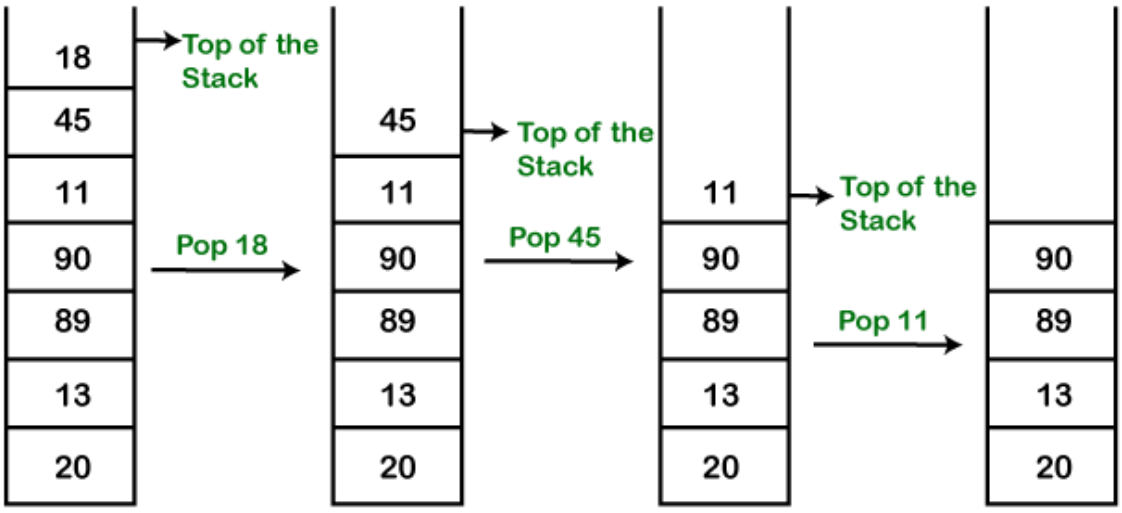


Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



Push operation

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



Pop operation

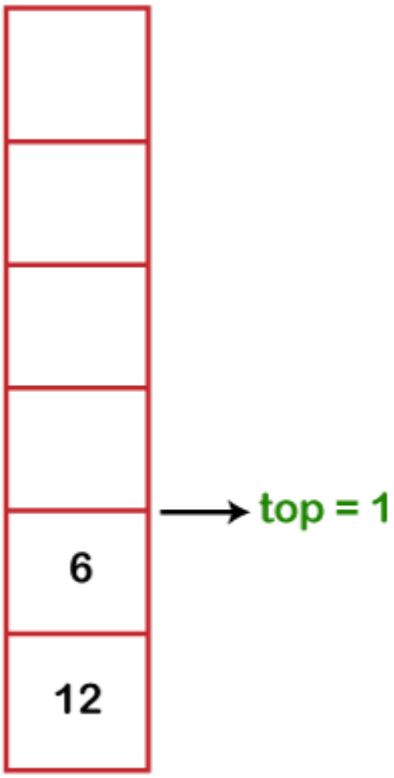
Empty Stack: If the stack has no element is known as an **empty stack**. When the stack is empty the value of the top variable is -1.

When we push an element into the stack the top is **increased by 1**. In the following figure,

- 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 the following figure, we have popped 9.

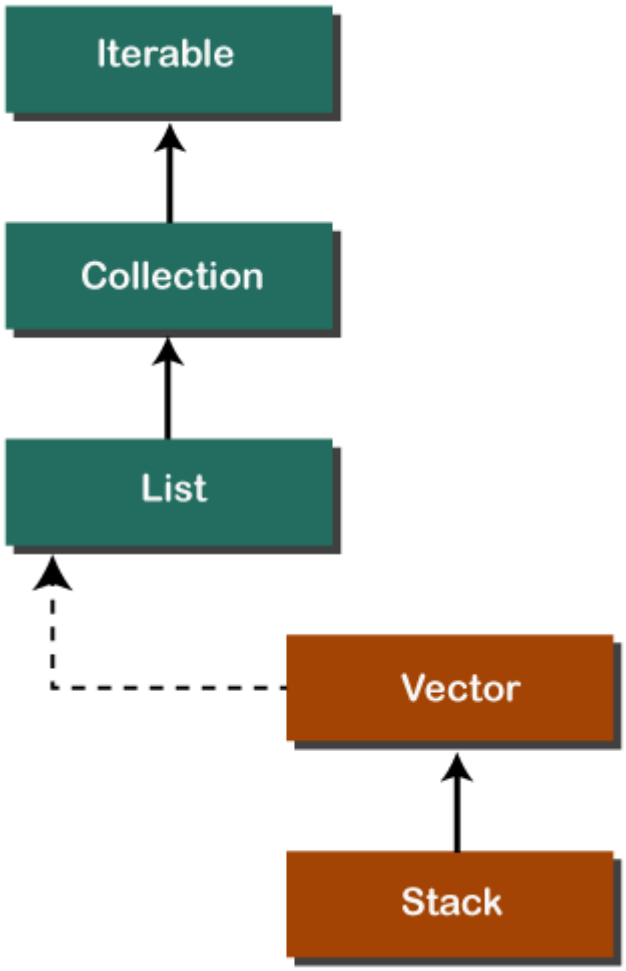


The following table shows the different values of the top.

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.

Java Stack Class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package. The stack class arranged in the Collections framework hierarchy, as shown below.



Stack Class Constructor

The Stack class contains only the **default constructor** that creates an empty stack.

1. **public** Stack()

Creating a Stack

If we want to create a stack, first, import the java.util package and create an object of the Stack class.

```
1. Stack stk = new Stack();
```

Or

```
1. Stack<type> stk = new Stack<>();
```

Where type denotes the type of stack like Integer, String, etc.

Methods of the Stack Class

We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the Java 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.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

Stack Class empty() Method

The **empty()** method of the Stack class check the stack is empty or not. If the stack is empty, it returns true, else returns false. We can also use the isEmpty() method of the Vector class.

Syntax

```
1. public boolean empty()
```

Returns: The method returns true if the stack is empty, else returns false.

In the following example, we have created an instance of the Stack class. After that, we have invoked the empty() method two times. The first time it returns **true** because we

have not pushed any element into the stack. After that, we have pushed elements into the stack. Again we have invoked the empty() method that returns **false** because the stack is not empty.

StackEmptyMethodExample.java

```
1. import java.util.Stack;
2. public class StackEmptyMethodExample
3. {
4. public static void main(String[] args)
5. {
6. //creating an instance of Stack class
7. Stack<Integer> stk= new Stack<>();
8. // checking stack is empty or not
9. boolean result = stk.empty();
10. System.out.println("Is the stack empty? " + result);
11. // pushing elements into stack
12. stk.push(78);
13. stk.push(113);
14. stk.push(90);
15. stk.push(120);
16. //prints elements of the stack
17. System.out.println("Elements in Stack: " + stk);
18. result = stk.empty();
19. System.out.println("Is the stack empty? " + result);
20. }
21. }
```

Output:

```
Is the stack empty? true
Elements in Stack: [78, 113, 90, 120]
Is the stack empty? false
```

Stack Class push() Method

The method inserts an item onto the top of the stack. It works the same as the method addElement(item) method of the Vector class. It passes a parameter **item** to be pushed into the stack.

Syntax

```
1. public E push(E item)
```

Parameter: An item to be pushed onto the top of the stack.

Returns: The method returns the argument that we have passed as a parameter.

Stack Class pop() Method

The method removes an object at the top of the stack and returns the same object. It throws **EmptyStackException** if the stack is empty.

Syntax

```
public E pop()
```

Returns: It returns an object that is at the top of the stack.

Let's implement the stack in a Java program and perform push and pop operations.

StackPushPopExample.java

```
import java.util.*;
public class StackPushPopExample
{
public static void main(String args[])
{
//creating an object of Stack class
Stack <Integer> stk = new Stack<>();
System.out.println("stack: " + stk);
//pushing elements into the stack
pushelmnt(stk, 20);
pushelmnt(stk, 13);
pushelmnt(stk, 89);
pushelmnt(stk, 90);
pushelmnt(stk, 11);
pushelmnt(stk, 45);
pushelmnt(stk, 18);
//popping elements from the stack
popelmnt(stk);
popelmnt(stk);
//throws exception if the stack is empty
try
{
popelmnt(stk);
}
catch (EmptyStackException e)
{
System.out.println("empty stack");
}
}
```

```
30. //performing push operation
31. static void pushelmnt(Stack stk, int x)
32. {
33. //invoking push() method
34. stk.push(new Integer(x));
35. System.out.println("push -> " + x);
36. //prints modified stack
37. System.out.println("stack: " + stk);
38. }
39. //performing pop operation
40. static void popelmnt(Stack stk)
41. {
42. System.out.print("pop -> ");
43. //invoking pop() method
44. Integer x = (Integer) stk.pop();
45. System.out.println(x);
46. //prints modified stack
47. System.out.println("stack: " + stk);
48. }
49. }
```

Output:

```
stack: []
push -> 20
stack: [20]
push -> 13
stack: [20, 13]
push -> 89
stack: [20, 13, 89]
push -> 90
stack: [20, 13, 89, 90]
push -> 11
stack: [20, 13, 89, 90, 11]
push -> 45
stack: [20, 13, 89, 90, 11, 45]
push -> 18
stack: [20, 13, 89, 90, 11, 45, 18]
pop -> 18
stack: [20, 13, 89, 90, 11, 45]
pop -> 45
stack: [20, 13, 89, 90, 11]
pop -> 11
stack: [20, 13, 89, 90]
```

Stack Class peek() Method

It looks at the element that is at the top in the stack. It also throws **EmptyStackException** if the stack is empty.

Syntax

```
1. public E peek()
```

Returns: It returns the top elements of the stack.

Let's see an example of the peek() method.

StackPeekMethodExample.java

```
1. import java.util.Stack;
2. public class StackPeekMethodExample
3. {
4. public static void main(String[] args)
5. {
6. Stack<String> stk= new Stack<>();
7. // pushing elements into Stack
8. stk.push("Apple");
9. stk.push("Grapes");
10. stk.push("Mango");
11. stk.push("Orange");
12. System.out.println("Stack: " + stk);
13. // Access element from the top of the stack
14. String fruits = stk.peek();
15. //prints stack
16. System.out.println("Element at top: " + fruits);
17. }
18. }
```

Output:

```
Stack: [Apple, Grapes, Mango, Orange]
Element at the top of the stack: Orange
```

Stack Class search() Method

The method searches the object in the stack from the top. It parses a parameter that we want to search for. It returns the 1-based location of the object in the stack. Thes topmost object of the stack is considered at distance 1.

Suppose, o is an object in the stack that we want to search for. The method returns the distance from the top of the stack of the occurrence nearest the top of the stack. It uses **equals()** method to search an object in the stack.

Syntax

```
1. public int search(Object o)
```

Parameter: o is the desired object to be searched.

Returns: It returns the object location from the top of the stack. If it returns -1, it means that the object is not on the stack.

Let's see an example of the search() method.

StackSearchMethodExample.java

```
1. import java.util.Stack;
2. public class StackSearchMethodExample
3. {
4. public static void main(String[] args)
5. {
6. Stack<String> stk= new Stack<>();
7. //pushing elements into Stack
8. stk.push("Mac Book");
9. stk.push("HP");
10. stk.push("DELL");
11. stk.push("Asus");
12. System.out.println("Stack: " + stk);
13. // Search an element
14. int location = stk.search("HP");
15. System.out.println("Location of Dell: " + location);
16. }
17. }
```

Java Stack Operations

Size of the Stack

We can also find the size of the stack using the size() method of the Vector class. It returns the total number of elements (size of the stack) in the stack.

Syntax

```
1. public int size()
```

Let's see an example of the size() method of the Vector class.

StackSizeExample.java

```
1. import java.util.Stack;
2. public class StackSizeExample
3. {
4. public static void main (String[] args)
5. {
6. Stack stk = new Stack();
```

```
7. stk.push(22);
8. stk.push(33);
9. stk.push(44);
10. stk.push(55);
11. stk.push(66);
12. // Checks the Stack is empty or not
13. boolean rslt=stk.empty();
14. System.out.println("Is the stack empty or not? " +rslt);
15. // Find the size of the Stack
16. int x=stk.size();
17. System.out.println("The stack size is: "+x);
18. }
19. }
```

Output:

```
Is the stack empty or not? false
The stack size is: 5
```

Iterate Elements

Iterate means to fetch the elements of the stack. We can fetch elements of the stack using three different methods are as follows:

- Using **iterator()** Method
- Using **forEach()** Method
- Using **listIterator()** Method

Using the iterator() Method

It is the method of the Iterator interface. It returns an iterator over the elements in the stack. Before using the iterator() method import the java.util.Iterator package.

Syntax

```
1. Iterator<T> iterator()
```

Let's perform an iteration over the stack.

StackIterationExample1.java

```
1. import java.util.Iterator;
2. import java.util.Stack;
3. public class StackIterationExample1
4. {
5. public static void main (String[] args)
6. {
```

```
7.  //creating an object of Stack class
8.  Stack stk = new Stack();
9.  //pushing elements into stack
10. stk.push("BMW");
11. stk.push("Audi");
12. stk.push("Ferrari");
13. stk.push("Bugatti");
14. stk.push("Jaguar");
15. //iteration over the stack
16. Iterator iterator = stk.iterator();
17. while(iterator.hasNext())
18. {
19. Object values = iterator.next();
20. System.out.println(values);
21. }
22. }
23. }
```

Output:

```
BMW
Audi
Ferrari
Bugatti
Jaguar
```

Using the forEach() Method

Java provides a forEach() method to iterate over the elements. The method is defined in the **Iterable** and **Stream** interface.

Syntax

```
1. default void forEach(Consumer<super T>action)
```

Let's iterate over the stack using the forEach() method.

StackIterationExample2.java

```
1. import java.util.*;
2. public class StackIterationExample2
3. {
4. public static void main (String[] args)
5. {
6. //creating an instance of Stack class
7. Stack <Integer> stk = new Stack<>();
8. //pushing elements into stack
```

```
9. stk.push(119);
10. stk.push(203);
11. stk.push(988);
12. System.out.println("Iteration over the stack using forEach() Method:");
13. //invoking forEach() method for iteration over the stack
14. stk.forEach(n ->
15. {
16. System.out.println(n);
17. });
18. }
19. }
```

Output:

```
Iteration over the stack using forEach() Method:
119
203
988
```

Using listIterator() 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.

Syntax

```
1. ListIterator listIterator(int index)
```

Parameter: The method parses a parameter named **index.**

Returns: This method returns a list iterator over the elements, in sequence.

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

Let's iterate over the stack using the listIterator() method.

StackIterationExample3.java

```
1. import java.util.Iterator;
2. import java.util.ListIterator;
3. import java.util.Stack;
4.
5. public class StackIterationExample3
6. {
7. public static void main (String[] args)
8. {
9. Stack <Integer> stk = new Stack<>();
```

```
10. stk.push(119);
11. stk.push(203);
12. stk.push(988);
13. ListIterator<Integer> ListIterator = stk.listIterator(stk.size());
14. System.out.println("Iteration over the Stack from top to bottom:");
15. while (ListIterator.hasPrevious())
16. {
17. Integer avg = ListIterator.previous();
18. System.out.println(avg);
19. }
20. }
21. }
```

Output:

```
Iteration over the Stack from top to bottom:
988
203
119
```

Java Vector

Vector is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-

- Vector is synchronized.
- Java Vector contains many legacy methods that are not the part of a collections framework.

Java Vector class Declaration

1. **public class** Vector<E>
2. **extends** Object<E>
3. **implements** List<E>, Cloneable, Serializable

Java Vector Constructors

Vector class supports four types of constructors. These are given below:

SN	Constructor	Description
1)	vector()	It constructs an empty vector with the default size as 10.
2)	vector(int initialCapacity)	It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
3)	vector(int initialCapacity, int capacityIncrement)	It constructs an empty vector with the specified initial capacity and capacity increment.
4)	Vector(Collection<? extends E> c)	It constructs a vector that contains the elements of a collection c.

Java Vector Methods

The following are the list of Vector class methods:

SN	Method	Description
1)	add()	It is used to append the specified element in the given vector.
2)	addAll()	It is used to append all of the elements in the specified collection to the end of this Vector.
3)	addElement()	It is used to append the specified component to the end of this vector. It increases the vector size by one.
4)	capacity()	It is used to get the current capacity of this vector.
5)	clear()	It is used to delete all of the elements from this vector.
6)	clone()	It returns a clone of this vector.
7)	contains()	It returns true if the vector contains the specified element.
8)	containsAll()	It returns true if the vector contains all of the elements in the specified collection.
9)	copyInto()	It is used to copy the components of the vector into the specified array.
10)	elementAt()	It is used to get the component at the specified index.
11)	elements()	It returns an enumeration of the components of a vector.
12)	ensureCapacity()	It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument.
13)	equals()	It is used to compare the specified object with the vector for equality.
14)	firstElement()	It is used to get the first component of the vector.
15)	forEach()	It is used to perform the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
16)	get()	It is used to get an element at the specified position in the vector.
17)	hashCode()	It is used to get the hash code value of a vector.

18)	indexOf()	It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
19)	insertElementAt()	It is used to insert the specified object as a component in the given vector at the specified index.
20)	isEmpty()	It is used to check if this vector has no components.
21)	iterator()	It is used to get an iterator over the elements in the list in proper sequence.
22)	lastElement()	It is used to get the last component of the vector.
23)	lastIndexOf()	It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
24)	listIterator()	It is used to get a list iterator over the elements in the list in proper sequence.
25)	remove()	It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged.
26)	removeAll()	It is used to delete all the elements from the vector that are present in the specified collection.
27)	removeAllElements()	It is used to remove all elements from the vector and set the size of the vector to zero.
28)	removeElement()	It is used to remove the first (lowest-indexed) occurrence of the argument from the vector.
29)	removeElementAt()	It is used to delete the component at the specified index.
30)	removeIf()	It is used to remove all of the elements of the collection that satisfy the given predicate.
31)	removeRange()	It is used to delete all of the elements from the vector whose index is between fromIndex, inclusive and toIndex, exclusive.
32)	replaceAll()	It is used to replace each element of the list with the result of applying the operator to that element.
33)	retainAll()	It is used to retain only that element in the vector which is contained in the specified collection.

34)	set()	It is used to replace the element at the specified position in the vector with the specified element.
35)	setElementAt()	It is used to set the component at the specified index of the vector to the specified object.
36)	setSize()	It is used to set the size of the given vector.
37)	size()	It is used to get the number of components in the given vector.
38)	sort()	It is used to sort the list according to the order induced by the specified Comparator.
39)	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the list.
40)	subList()	It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive.
41)	toArray()	It is used to get an array containing all of the elements in this vector in correct order.
42)	toString()	It is used to get a string representation of the vector.
43)	trimToSize()	It is used to trim the capacity of the vector to the vector's current size.

Java Vector Example

```
1. import java.util.*;
2. public class VectorExample {
3.     public static void main(String args[]) {
4.       //Create a vector
5.        Vector<String> vec = new Vector<String>();
6.       //Adding elements using add() method of List
7.        vec.add("Tiger");
8.       vec.add("Lion");
9.        vec.add("Dog");
10.      vec.add("Elephant");
11.       //Adding elements using addElement() method of Vector
12.      vec.addElement("Rat");
13.       vec.addElement("Cat");
14.      vec.addElement("Deer");
15.
```

```
16.         System.out.println("Elements are: "+vec);
17.       }
18. }
```

Test it Now

Output:

```
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
```

Java Vector Example 2

```
1.  import java.util.*;
2.  public class VectorExample1 {
3.        public static void main(String args[]) {
4.          //Create an empty vector with initial capacity 4
5.           Vector<String> vec = new Vector<String>(4);
6.          //Adding elements to a vector
7.           vec.add("Tiger");
8.          vec.add("Lion");
9.           vec.add("Dog");
10.         vec.add("Elephant");
11.          //Check size and capacity
12.         System.out.println("Size is: "+vec.size());
13.          System.out.println("Default capacity is: "+vec.capacity());
14.         //Display Vector elements
15.          System.out.println("Vector element is: "+vec);
16.         vec.addElement("Rat");
17.          vec.addElement("Cat");
18.         vec.addElement("Deer");
19.          //Again check size and capacity after two insertions
20.         System.out.println("Size after addition: "+vec.size());
21.          System.out.println("Capacity after addition is: "+vec.capacity());
22.         //Display Vector elements again
23.          System.out.println("Elements are: "+vec);
24.         //Checking if Tiger is present or not in this vector
25.           if(vec.contains("Tiger"))
26.          {
27.              System.out.println("Tiger is present at the index " +vec.indexOf("Tige
    r"));
28.          }
29.           else
30.          {
31.              System.out.println("Tiger is not present in the list.");
```

```
32.          }
33.           //Get the first element
34.          System.out.println("The first animal of the vector is = "+vec.firstElement());
35.           //Get the last element
36.          System.out.println("The last animal of the vector is = "+vec.lastElement());
37.       }
38. }
```

Test it Now

Output:

```
Size is: 4
Default capacity is: 4
Vector element is: [Tiger, Lion, Dog, Elephant]
Size after addition: 7
Capacity after addition is: 8
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
Tiger is present at the index 0
The first animal of the vector is = Tiger
The last animal of the vector is = Deer
```

Java Vector Example 3

```
1.  import java.util.*;
2.  public class VectorExample2 {
3.        public static void main(String args[]) {
4.        //Create an empty Vector
5.         Vector<Integer> in = new Vector<>();
6.        //Add elements in the vector
7.         in.add(100);
8.        in.add(200);
9.         in.add(300);
10.       in.add(200);
11.        in.add(400);
12.       in.add(500);
13.        in.add(600);
14.       in.add(700);
15.        //Display the vector elements
16.       System.out.println("Values in vector: " +in);
17.        //use remove() method to delete the first occurence of an element
18.       System.out.println("Remove first occourence of element 200: "+in.remove((Integ
    er)200));
19.        //Display the vector elements afre remove() method
20.       System.out.println("Values in vector: " +in);
21.        //Remove the element at index 4
22.       System.out.println("Remove element at index 4: " +in.remove(4));
23.        System.out.println("New Value list in vector: " +in);
```

```
24.         //Remove an element
25.         in.removeElementAt(5);
26.         //Checking vector and displays the element
27.         System.out.println("Vector element after removal: " +in);
28.         //Get the hashcode for this vector
29.         System.out.println("Hash code of this vector = "+in.hashCode());
30.         //Get the element at specified index
31.         System.out.println("Element at index 1 is = "+in.get(1));
32.          }
33. }
```

Test it Now

Output:

```
Values in vector: [100, 200, 300, 200, 400, 500, 600, 700]
Remove first occourence of element 200: true
Values in vector: [100, 300, 200, 400, 500, 600, 700]
Remove element at index 4: 500
New Value list in vector: [100, 300, 200, 400, 600, 700]
Vector element after removal: [100, 300, 200, 400, 600]
Hash code of this vector = 130123751
Element at index 1 is = 300
```