Java List Interface

Last Updated : 21 Jan, 2025

The List Interface in Java extends the Collection Interface and is a part of java.util package. It is used to store the **ordered collections of elements**. So in a Java List, you can organize and manage the data sequentially.

- Maintained the order of elements in which they are added.
- Allows the duplicate elements.
- The implementation classes of the List interface are ArrayList, LinkedList, Stack, and Vector.
- Can add Null values that depend on the implementation.
- The List interface offers methods to access elements by their index and includes the **listIterator() method**, which returns a **ListIterator**.
- Using ListIterator, we can traverse the list in both forward and backward directions.

Example:

```java
// Java program to show the use of List Interface
import java.util.*;

class GFG {

    public static void main(String[] args)
    {

        // Creating a List of Strings using ArrayList
        List<String> li = new ArrayList<>();

        // Adding elements in List
        li.add("Java");
        li.add("Python");
        li.add("DSA");
        li.add("C++");
        System.out.println("Elements of List are:");

        // Iterating through the list
        for (String s : li) {
            System.out.println(s);
        }

        // Accessing elements
        System.out.println("Element at Index 1: "+ li.get(1));

        // Updating elements
        li.set(1, "JavaScript");
        System.out.println("Updated List: " + li);

        // Removing elements
        li.remove("C++");
        System.out.println("List After Removing Element: " + li);
    }
}
```

Output

```
Elements of List are:
Java
Python
DSA
C++
Element at Index 1: Python
Updated List: [Java, JavaScript, DSA, C++]
List After Removing Element: [Java, JavaScript, DSA]
```

Declaration of Java List Interface

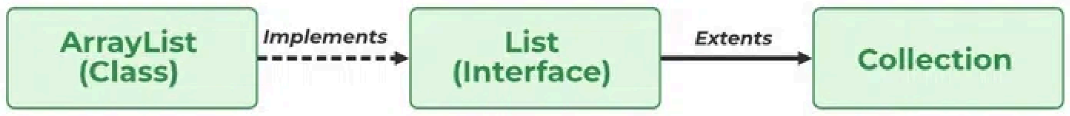
```
public interface List<E> extends Collection<E> ;
```

Let us elaborate on creating objects or instances in a List. Since **List** is an interface, objects cannot be created of the type list. We always need a class that implements this **List** in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List. Just like several other user-defined 'interfaces' implemented by user-defined 'classes', **List** is an 'interface', implemented by the **ArrayList** class, pre-defined in **java.util** package.

Syntax:

```
List<Obj> list = new ArrayList<Obj> ();
```

Obj is the type of the object to be stored in List.



The common implementation classes of the `List` interface are **ArrayList**, **LinkedList**, **Stack**, and **Vector**:

- **ArrayList** and **LinkedList** are the most widely used due to their dynamic resizing and efficient performance for specific operations.
- **Stack** is a subclass of `Vector`, designed for Last-In-First-Out (LIFO) operations.
- **Vector** is considered a legacy class and is rarely used in modern Java programming. It is replaced by ArrayList and java.util.concurrent package.

Now let us perform various operations using List Interface to have a [illegible] operations listed below and later on implementing them via clean Java codes.

Java Course Java Arrays Java Strings Java OOPs Java Collection Java 8 Tutorial Java Multithrea

Java List – Operations

Since List is an interface, it can be used only with a class that implements this interface. Now, let's see how to perform a few frequently used operations on the List.

- **Operation 1:** Adding elements to List using add() method
- **Operation 2:** Updating elements in List using set() method

- **Operation 3:** Searching for elements using indexOf(), lastIndexOf methods
- **Operation 4:** Removing elements using remove() method
- **Operation 5:** Accessing Elements in List using get() method
- **Operation 6:** Checking if an element is present in the List using contains() method

Now let us discuss the operations individually and implement the same in the code to grasp a better grip over it.

1. Adding Elements

In order to add an element to the list, we can use the **add()** method. This method is overloaded to perform multiple operations based on different parameters.

Parameters: It takes 2 parameters, namely:

- **add(Object o):** This method is used to add an element at the end of the List.
- **add(int index, Object o):** This method is used to add an element at a specific index in the List

Note: If we do not specify the length of the array in the ArrayList constructor while creating the List object, using add(int index, Object) for any index i will throw an Exception if we have not specified the values for 0 to i-1 index already.

Example:

```java
// Java Program to Add Elements to a List
import java.util.*;

class GFG {

    public static void main(String args[])
    {
        // Creating an object of List interface,
        // implemented by ArrayList class
        List<String> al = new ArrayList<>();

        // Adding elements to object of List interface
        // Custom elements
        al.add("Geeks");
        al.add("Geeks");
        al.add(1, "For");

        // Print all the elements inside the
        // List interface object
        System.out.println(al);
    }
}
```

Output

```
[Geeks, For, Geeks]
```

Note: If we try to add element at index 1 before adding elements at index 0 it will throw an error. It is always recommended to add elements in a particular index only when the size is defined or to add them sequentially.

2. Updating Elements

After adding the elements, if we wish to change the element, it can be done using the **set()** method. Since List is indexed, the element which we wish to change is referenced by the index of the element. Therefore, this method takes an index and the updated element which needs to be inserted at that index.

Example:

```
// Java Program to Update Elements in a List
import java.util.*;

class GFG {

    public static void main(String args[])
    {
        // Creating an object of List interface
        List<String> al = new ArrayList<>();

        // Adding elements to object of List class
        al.add("Geeks");
        al.add("Geeks");
        al.add(1, "Geeks");

        // Display theinitial elements in List
        System.out.println("Initial ArrayList " + al);

        // Setting (updating) element at 1st index
        // using set() method
        al.set(1, "For");
        // Print and display the updated List
        System.out.println("Updated ArrayList " + al);
    }
}
```

Output

```
Initial ArrayList [Geeks, Geeks, Geeks]
Updated ArrayList [Geeks, For, Geeks]
```

3. Searching Elements

Searching for elements in the List interface is a common operation in Java programming. The List interface provides several methods to search for elements, such as the indexOf(), lastIndexOf() methods.

The indexOf() method returns the index of the first occurrence of a specified element in the list, while the lastIndexOf() method returns the index of the last occurrence of a specified element.

Parameters:

- **indexOf(Object o):** Returns the index of the first occurrence of the specified element in the list, or -1 if the element is not found
- **lastIndexOf(Object o):** Returns the index of the last occurrence of the specified element in the list, or -1 if the element is not found

Example:

```
// Java program to search the elements in a List
import java.util.*;

class GFG {

    public static void main(String[] args)
    {
        // create a list of integers
        List<Integer> al = new ArrayList<>();

        // add some integers to the list
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(2);

        // use indexOf() to find the first occurrence of an
        // element in the list
        int i = al.indexOf(2);

        System.out.println("First Occurrence of 2 is at Index: "+i);

        // use lastIndexOf() to find the last occurrence of
        // an element in the list
        int l = al.lastIndexOf(2);

        System.out.println("Last Occurrence of 2 is at Index: "+l);
    }
}
```

Output

```
First Occurrence of 2 is at Index: 1
Last Occurrence of 2 is at Index: 3
```

4. Removing Elements

In order to remove an element from a list, we can use the **remove()** method. This method is overloaded to perform multiple operations based on different parameters. They are:

Parameters:

- **remove(Object o):** This method is used to simply remove an object from the List. If there are multiple such objects, then the first occurrence of the object is removed.
- **remove(int index):** Since a List is indexed, this method takes an integer value which simply removes the element present at that specific index in the List. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated.

Example:

```
// Java Program to Remove Elements from a List
import java.util.ArrayList;
import java.util.List;

class GFG {

    public static void main(String args[])
    {
        // Creating List class object
        List<String> al = new ArrayList<>();

        // Adding elements to the object
        // Custom inputs
        al.add("Geeks");
        al.add("Geeks");

        // Adding For at 1st indexes
        al.add(1, "For");

        // Print the initialArrayList
        System.out.println("Initial ArrayList " + al);

        // Now remove element from the above list
        // present at 1st index
        al.remove(1);

        // Print the List after removal of element
        System.out.println("After the Index Removal " + al);

        // Now remove the current object from the updated
        // List
        al.remove("Geeks");

        // Finally print the updated List now
        System.out.println("After the Object Removal " + al);
    }
}
```

Output

```
Initial ArrayList [Geeks, For, Geeks]
After the Index Removal [Geeks, Geeks]
After the Object Removal [Geeks]
```

5. Accessing Elements

In order to access an element in the list, we can use the **get()** method, which returns the element at the specified index

Parameters:

- get(int index): This method returns the element at the specified index in the list.

Example:

```
// Java Program to Access Elements of a List
import java.util.*;
```

```
class GFG {

    public static void main(String args[])
    {
        // Creating an object of List interface,
        // implemented by ArrayList class
        List<String> al = new ArrayList<>();

        // Adding elements to object of List interface
        al.add("Geeks");
        al.add("For");
        al.add("Geeks");

        // Accessing elements using get() method
        String first = al.get(0);
        String second = al.get(1);
        String third = al.get(2);

        // Printing all the elements inside the
        // List interface object
        System.out.println(first);
        System.out.println(second);
        System.out.println(third);
        System.out.println(al);
    }
}
```

Output

```
Geeks
For
Geeks
[Geeks, For, Geeks]
```

6. Checking if an element is present or not

In order to check if an element is present in the list, we can use the contains() method. This method returns true if the specified element is present in the list, otherwise, it returns false.

Parameters:

- contains(Object o): This method takes a single parameter, the object to be checked if it is present in the list.

Example:

```java
// Java Program to Check if an Element is Present in a List
import java.util.*;

class GFG {

    public static void main(String args[])
    {
        // Creating an object of List interface,
        // implemented by ArrayList class
        List<String> al = new ArrayList<>();

        // Adding elements to object of List interface
        al.add("Geeks");
        al.add("For");
        al.add("Geeks");

        // Checking if element is present using contains()
        // method
        boolean isPresent = al.contains("Geeks");

        // Printing the result
        System.out.println("Is Geeks present in the list? "+ isPresent);
    }
}
```

Output

```
Is Geeks present in the list? true
```

Complexity of List Interface in Java

Operation	Time Complexity	Space Complexity
Adding Element in List Interface	O(1)	O(1)
Remove Element from List Interface	O(N)	O(N)
Replace Element in List Interface	O(N)	O(N)
Traversing List Interface	O(N)	O(N)

Iterating over List Interface in Java

Till now we are having a very small input size and we are doing operations manually for every entity. Now let us discuss various ways by which we can iterate over the list to get them working for a larger sample set.

Methods: There are multiple ways to iterate through the List. The most famous ways are by using the basic for loop in combination with a get() method to get the element at a specific index and the advanced for a loop.

Example:

```java
// Java program to Iterate the Elements
// in an List
import java.util.*;

public class GFG {

    public static void main(String args[])
    {
        // Creating an empty Arraylist of string type
        List<String> al = new ArrayList<>();

        // Adding elements to above object of ArrayList
        al.add("Geeks");
        al.add("Geeks");

        // Adding element at specified position
        // inside list object
        al.add(1, "For");

        // Using  for loop for iteration
        for (int i = 0; i < al.size(); i++) {

            // Using get() method to
            // access particular element
            System.out.print(al.get(i) + " ");
        }

        // New line for better readability
        System.out.println();

        // Using for-each loop for iteration
        for (String str : al)

            // Printing all the elements
            // which was inside object
            System.out.print(str + " ");
    }
}
```

Output

```
Geeks For Geeks
Geeks For Geeks
```

Methods of the List Interface

Since the main concept behind the different types of lists is the same, the list interface contains the following methods:

Method	Description
add(int index, element)	This method is used with Java List Interface to add an element at a particular index in the list. When a single parameter is passed, it simply adds the element at the end of the list.
addAll(int index, Collection collection)	This method is used with List Interface in Java to add all the elements in the given collection to the list. When a single parameter is passed, it adds all the elements of the given collection at the end of the list.
size()	This method is used with Java List Interface to return the size of the list.
clear()	This method is used to remove all the elements in the list. However, the reference of the list created is still stored.
remove(int index)	This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
remove(element)	This method is used with Java List Interface to remove the first occurrence of the given element in the list.

Method	Description
get(int index)	This method returns elements at the specified index.
set(int index, element)	This method replaces elements at a given index with the new element. This function returns the element which was just replaced by a new element.
indexOf(element)	This method returns the first occurrence of the given element or *-1* if the element is not present in the list.
lastIndexOf(element)	This method returns the last occurrence of the given element or *-1* if the element is not present in the list.
equals(element)	This method is used with Java List Interface to compare the equality of the given element with the elements of the list.
hashCode()	This method is used with List Interface in Java to return the hashcode value of the given list.
isEmpty()	This method is used with Java List Interface to check if the list is empty or not. It returns true if the list is empty, else false.
contains(element)	This method is used with List Interface in Java to check if the list contains the given element or not. It returns true if the list contains the element.
containsAll(Collection collection)	This method is used with Java List Interface to check if the list contains all the collection of elements.

Method	Description
sort(Comparator comp)	This method is used with List Interface in Java to sort the elements of the list on the basis of the given comparator.

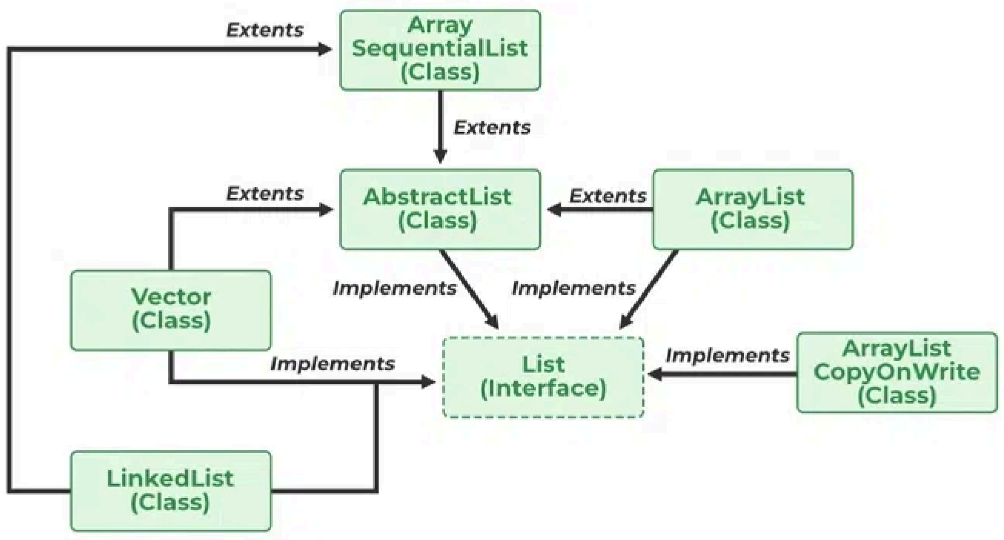
Java List vs Set

Both the List interface and the Set interface inherits the Collection interface. However, there exists some differences between them.

List	Set
The List is an ordered sequence.	The Set is an unordered sequence.
List allows duplicate elements	Set doesn't allow duplicate elements.
Elements by their position can be accessed.	Position access to elements is not allowed.
Multiple null elements can be stored.	The null element can store only once.
List implementations are ArrayList, LinkedList, Vector, Stack	Set implementations are HashSet, LinkedHashSet.

Classes Association with a Java List Interface

Now let us discuss the classes that implement the List Interface for which first do refer to the pictorial representation below to have a better understanding of the List interface. It is as follows:



AbstractList, CopyOnWriteArrayList, and the AbstractSequentialList are the classes that implement the List interface. A separate functionality is implemented in each of the mentioned classes. They are as follows:

1. **AbstractList:** This class is used to implement an unmodifiable list, for which one needs to only extend this AbstractList Class and implement only the *get()* and the *size()* methods.
2. **CopyOnWriteArrayList:** This class implements the list interface. It is an enhanced version of ArrayList in which all the modifications(add, set, remove, etc.) are implemented by making a fresh copy of the list.
3. **AbstractSequentialList:** This class implements the Collection interface and the AbstractCollection class. This class is used to implement an unmodifiable list, for which one needs to only extend this AbstractList Class and implement only the *get()* and the *size()* methods.

We will proceed in this manner.

- ArrayList
- Vector
- Stack
- LinkedList

Let us discuss them sequentially and implement the same to figure out the working of the classes with the List interface.

1. ArrayList

An ArrayList class which is implemented in the collection framework provides us with dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. Let’s see how to create a list object using this class.

Example:

```java
// Java program to demonstrate the
// creation of list object using the
// ArrayList class

import java.io.*;
import java.util.*;

class GFG {
    public static void main(String[] args)
    {
        // Size of ArrayList
        int n = 5;

        // Declaring the List with initial size n
        List<Integer> arrli = new ArrayList<Integer>(n);

        // Appending the new elements
        // at the end of the list
        for (int i = 1; i <= n; i++)
            arrli.add(i);

        // Printing elements
        System.out.println(arrli);

        // Remove element at index 3
        arrli.remove(3);

        // Displaying the list after deletion
        System.out.println(arrli);

        // Printing elements one by one
        for (int i = 0; i < arrli.size(); i++)
            System.out.print(arrli.get(i) + " ");
    }
}
```

Output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

2. Vector

Vector is a class that is implemented in the collection framework implements a growable array of objects. Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index. Vectors basically fall in legacy classes but now it is fully compatible with collections. Let's see how to create a list object using this class.

Example:

```java
// Java program to demonstrate the
// creation of list object using the
// Vector class

import java.io.*;
import java.util.*;

class GFG {
    public static void main(String[] args)
    {
        // Size of the vector
        int n = 5;

        // Declaring the List with initial size n
        List<Integer> v = new Vector<Integer>(n);

        // Appending the new elements
        // at the end of the list
        for (int i = 1; i <= n; i++)
            v.add(i);

        // Printing elements
        System.out.println(v);

        // Remove element at index 3
        v.remove(3);

        // Displaying the list after deletion
        System.out.println(v);

        // Printing elements one by one
        for (int i = 0; i < v.size(); i++)
            System.out.print(v.get(i) + " ");
    }
}
```

Output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

3. Stack

Stack is a class that is implemented in the collection framework and extends the vector class models and implements the Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek. Let's see how to create a list object using this class.

Example:

```java
// Java program to demonstrate the
// creation of list object using the
// Stack class

import java.io.*;
import java.util.*;

class GFG {
    public static void main(String[] args)
    {
        // Size of the stack
        int n = 5;

        // Declaring the List
        List<Integer> s = new Stack<Integer>();

        // Appending the new elements
        // at the end of the list
        for (int i = 1; i <= n; i++)
            s.add(i);

        // Printing elements
        System.out.println(s);

        // Remove element at index 3
        s.remove(3);

        // Displaying the list after deletion
        System.out.println(s);

        // Printing elements one by one
        for (int i = 0; i < s.size(); i++)
            System.out.print(s.get(i) + " ");
    }
}
```

Output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
```

```
1 2 3 5
```

4. LinkedList

LinkedList is a class that is implemented in the collection framework which inherently implements the linked list data structure. It is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays. Let's see how to create a list object using this class.

Example:

```java
// Java program to demonstrate the
// creation of list object using the
// LinkedList class

import java.io.*;
import java.util.*;

class GFG {
    public static void main(String[] args)
    {
        // Size of the LinkedList
        int n = 5;

        // Declaring the List with initial size n
        List<Integer> ll = new LinkedList<Integer>();

        // Appending the new elements
        // at the end of the list
        for (int i = 1; i <= n; i++)
            ll.add(i);

        // Printing elements
        System.out.println(ll);

        // Remove element at index 3
        ll.remove(3);

        // Displaying the list after deletion
        System.out.println(ll);

        // Printing elements one by one
        for (int i = 0; i < ll.size(); i++)
            System.out.print(ll.get(i) + " ");
    }
}
```

Output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Comment More info

Placement Training Program

Next Article

ArrayList in Java

Similar Reads

Why to Use Comparator Interface Rather than Comparable Interface...

In Java, the Comparable and Comparator interfaces are used to sort collections of objects based on certain criteria. The Comparable interface...

8 min read

How to Test Java List Interface Methods using Mockito?

Mockito is an open-source testing framework used for unit testing of Java applications. It plays a vital role in developing testable applications....

5 min read

Java 8 | DoubleToIntFunction Interface in Java with Example

The DoubleToIntFunction Interface is a part of the java.util.function package which has been introduced since Java 8, to implement functiona...

1 min read