**List interface Overview**

* An ordered collection (also known as a sequence). It guarantees an insertion order.
* The user of this interface has precise control over where in the list each element is inserted.
* It guarantees insertion order so a user can access elements by their integer index (position in the list), and search for elements in the list.

***ArrayList* class Overview**

Java *ArrayList* class uses a dynamic array for storing the elements. It inherits *AbstractList* class and implements *List* interface.

The important points about Java *ArrayList*class are:

* Java *ArrayList*class can contain duplicate elements.
* Java *ArrayList*class maintains insertion order.
* Java *ArrayList*class is non-synchronized.
* Java *ArrayList*allows random access because array works at the index basis.
* In Java *ArrayList*class, manipulation is slow because a lot of shifting needs to have occurred if any element is removed from the array list.

**Demonstration of how the *List* contains duplicates and null values?**

This example demonstrates how the *ArrayList*contains duplicate and null values.

In this example, we have passed 2 null values and output should be null values. Note that the output of this example print duplicate elements.

public class ListInterfaceArrayListImpl {

public static void main(String[] args) {

nullValueDemo();

duplicateValueDemo();

}

private static void nullValueDemo() {

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

list.add(null);

list.add(null);

System.out.println(list.toString());

}

private static void duplicateValueDemo() {

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

list.add("duplicate");

list.add("duplicate");

System.out.println(list.toString());

}

}

Output :

[null, null]

[duplicate, duplicate]

***List*positional access and search operations**

Manipulates elements based on their numerical position in the list. This includes methods such as *get, set, add, addAll, and remove*.

The basic positional access methods *get, set, add and remove*. (The set and remove operations return the old value that is being overwritten or removed.) Other operations (indexOf and lastIndexOf) return the first or last index of the specified element in the list.

Refer comments in source code are self descriptive.

// The basic positional access operations are get, set, add and remove.

private static void positionalAccess() {

List<String> list = new LinkedList<>();

list.add("element 1");

list.add("element 2");

list.add("element 3");

list.add("element 4");

// Replaces the element at the specified position in this list with

// the specified element (optional operation).

list.set(3, "element 5");

list.forEach( str -> System.out.println(" set element 5 --" + str));

// Inserts the specified element at the specified position in this list

// (optional operation).

// Shifts the element currently at that position (if any) and any

// subsequent

// elements to the right (adds one to their indices).

list.add(3, "element 4");

list.forEach( str -> System.out.println(" set element 5 --" + str));

// Returns the element at the specified position in this list.

System.out.println(list.get(0));

// Removes the element at the specified position in this list (optional

// operation).

// Shifts any subsequent elements to the left (subtracts one from their

// indices).//

// Returns the element that was removed from the list.

list.remove(1);

}

**How to search elements in *List*?**

*indexOf()*and *lastIndexOf()* methods can use to search an element in the list with specific index.

Please refer comments in source code are self descriptive.

private static void searchListDemo() {

List<String> searchList = new ArrayList();

searchList.add("element 1");

searchList.add("element 2");

searchList.add("element 3");

searchList.add("element 4");

// Returns the index of the first occurrence of the specified element in

// this list,

// or -1 if this list does not contain the element.

int index = searchList.indexOf("element 2");

System.out.println(" search element at index 0 --->" + index);

// Returns the index of the last occurrence of the specified element in

// this list,

// or -1 if this list does not contain the element

int lastIndex = searchList.lastIndexOf("element 2");

System.out.println(" search element at lastIndex 0 --->" + lastIndex);

}

**How to perform range view operations in *List*?**

The range-view operation, *subList(int fromIndex, int toIndex)*, returns a *List*view of the portion of this list whose indices range from fromIndex, inclusive, to toIndex, exclusive.

//Returns a view of the portion of this list between the specified fromIndex,

//inclusive, and toIndex, exclusive.

public void rangeViewDemo(){

List<String> list = new LinkedList<>();

list.add("element 1");

list.add("element 2");

list.add("element 3");

list.add("element 4");

//If fromIndex and toIndex are equal, the returned list is empty.)

for(String str : list.subList(0, 0)){

System.out.println(" sub list demo --" + str);

}

for(String str : list.subList(0, 1)){

System.out.println(" sub list demo --" + str);

}

}

**How to perform bulk operations using *List*?**

private static void bulkOperationDemo() {

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

list.add("element 1");

list.add("element 2");

list.add("element 3");

list.add("element 4");

// addAll() - Appends all of the elements in the specified collection to

// the end of this list,

// in the order that they are returned by the specified collection's

// iterator (optional operation).

List<String> union = new ArrayList<>();

union.addAll(list);

printMessage(union, "addALL operation example ");

// Retains only the elements in this list that are contained in

// the specified collection (optional operation).

List<String> intersection = new ArrayList<>();

intersection.add("element 1");

intersection.add("element 2");

intersection.add("element 3");

intersection.add("element 4");

System.out.println("retainAll -- > " + intersection.retainAll(list));

// Removes from this list all of its elements that are

// contained in the specified collection (optional operation).

List<String> difference = new ArrayList<>();

difference.add("element 1");

difference.add("element 2");

difference.add("element 3");

difference.add("element 4");

System.out.println("removeAll operation example ---> " + difference.removeAll(list));

printMessage(difference, "removeAll operation example ");

List<String> checking = new ArrayList<>();

checking.add("element 1");

checking.add("element 2");

checking.add("element 3");

checking.add("element 4");

System.out.println("containsAll operation example ---- > " + checking.containsAll(list));

}

private static void printMessage(List<String> list, String message) {

list.forEach(key -> System.out.println(message + key));

}

Output :

addALL operation example element 1

addALL operation example element 2

addALL operation example element 3

addALL operation example element 4

retainAll -- > false

removeAll operation example ---> true

containsAll operation example ---- > true

**Iterating over an *ArrayList***

// Three ways to iterator list

private static void iterateDemo() {

List<String> list = new LinkedList<>();

list.add("element 1");

list.add("element 2");

list.add("element 3");

list.add("element 4");

// using Iterator

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String str = iterator.next();

System.out.println(" only forward direction ---" + str);

}

// Using advanced for loop

for (String str : list) {

System.out.println(" only forward direction ---" + str);

}

// Java 8

list.forEach(str -> System.out.println(" only forward direction ---" + str));

}

**Traverse the list in either direction using *ListIterator*interface**

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A *ListIterator*has no current element; its cursor position always lies between the element that would be returned by a call to *previous()*and the element that would be returned by a call to *next()*.

// listOperation example

private static void listIterateDemo() {

List<String> list = new LinkedList<>();

list.add("element 1");

list.add("element 2");

list.add("element 3");

list.add("element 4");

ListIterator<String> iterator = list.listIterator();

while (iterator.hasNext()) {

String str = iterator.next();

System.out.println("forward direction ---" + str);

}

while (iterator.hasPrevious()) {

String str = iterator.previous();

System.out.println("backward direction ---" + str);

}

for (ListIterator<String> it = list.listIterator(list.size()); it.hasPrevious();) {

String t = it.previous();

System.out.println(t);

}

}

**Creating an *ArrayList*from another collection**

Let's see how to create an ArrayList from another collection using the *ArrayList(Collection c)* constructor.

How to add all the elements from an existing collection to the new *ArrayList* using the *addAll()*method.

import java.util.ArrayList;

import java.util.List;

public class CreateArrayListFromCollectionExample {

public static void main(String[] args) {

List<Integer> firstFivePrimeNumbers = new ArrayList<>();

firstFivePrimeNumbers.add(2);

firstFivePrimeNumbers.add(3);

firstFivePrimeNumbers.add(5);

firstFivePrimeNumbers.add(7);

firstFivePrimeNumbers.add(11);

// Creating an ArrayList from another collection

List<Integer> firstTenPrimeNumbers = new ArrayList<>(firstFivePrimeNumbers);

List<Integer> nextFivePrimeNumbers = new ArrayList<>();

nextFivePrimeNumbers.add(13);

nextFivePrimeNumbers.add(17);

nextFivePrimeNumbers.add(19);

nextFivePrimeNumbers.add(23);

nextFivePrimeNumbers.add(29);

// Adding an entire collection to an ArrayList

firstTenPrimeNumbers.addAll(nextFivePrimeNumbers);

System.out.println(firstTenPrimeNumbers);

}

}

Output

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

**Removing elements from an *ArrayList***

*ArrayList*class provides many methods to remove elements from *List*based on requirement. Let's discuss all the remove methods that *ArrayList*offers.

***remove(int index)***

Removes the element at the specified position in this list (optional operation).

List<String> programmingLanguages = new ArrayList<>();

programmingLanguages.add("C");

programmingLanguages.add("C++");

programmingLanguages.add("Java");

programmingLanguages.add("Kotlin");

// Remove the element at index `5`

programmingLanguages.remove(5);

System.out.println("After remove(5): " + programmingLanguages);

**remove(Object o)**

Use this method to removes the first occurrence of the specified element from this list, if it is present (optional operation).

List<String> programmingLanguages = new ArrayList<>();

programmingLanguages.add("C");

programmingLanguages.add("C++");

programmingLanguages.add("Java");

programmingLanguages.add("Kotlin");

boolean isRemoved = programmingLanguages.remove("Kotlin");

***removeAll()***

Removes from this list all of its elements that are contained in the specified collection (optional operation).

List<String> programmingLanguages = new ArrayList<>();

programmingLanguages.add("C");

programmingLanguages.add("C++");

programmingLanguages.add("Java");

programmingLanguages.add("Kotlin");

// Remove all the elements that exist in a given collection

List<String> scriptingLanguages = new ArrayList<>();

scriptingLanguages.add("Java");

scriptingLanguages.add("Kotlin");

programmingLanguages.removeAll(scriptingLanguages);

***removeIf()***

Removes all of the elements of this collection that satisfy the given predicate. Errors or runtime exceptions thrown during iteration or by the predicate are relayed to the caller.

List<String> programmingLanguages = new ArrayList<>();

programmingLanguages.add("C");

programmingLanguages.add("C++");

programmingLanguages.add("Java");

programmingLanguages.add("Kotlin");

// Remove all the elements that satisfy the given predicate

programmingLanguages.removeIf(new Predicate<String>() {

@Override

public boolean test(String s) {

return s.startsWith("C");

}

});

***clear()***

Removes all of the elements from this list (optional operation). The list will be empty after this call returns.

List<String> programmingLanguages = new ArrayList<>();

programmingLanguages.add("C");

programmingLanguages.add("C++");

programmingLanguages.add("Java");

programmingLanguages.add("Kotlin");

// Remove all elements from the ArrayList

programmingLanguages.clear();

System.out.println("After clear(): " + programmingLanguages);