AbstractList<E> class

AbstractList<E> class is abstraction layer for [List<E>](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-5list-interface.html) interface. Click here to learn [List<E>](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-5list-interface.html) interface. To learn how to iterate over ArrayList<E> click [here](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-6iterating-over.html).

Now let us start understanding AbstractList<E> class and its methods. This class is abstract so few methods does not have concrete implementation as those methods are specific to implementing classes that extends Abstract class and/or implement interface.

**Code for this class is referred and take from Javadocs.**

AbstractList<E> class definition

AbstractList<E> is abstract class and it extends [AbstractCollection<E>](http://data-structure-learning.blogspot.com/2015/05/java-collection-part-10.html) class. Also it implements [List<E>](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-5list-interface.html) interface.

**public** **abstract** **class** AbstractList<E> **extends** AbstractCollection<E> **implements** List<E>

Constructor for AbstractList<E> class.

Constructor of this class is protected by definition. And why is that? The reason is that the invocation can be done by subclass constructors only.

**protected** AbstractList() {

}

add(E e) and add(int index, E e)

add(E e) Appends the element specified in parameter to the end of the list. This method calls overloaded version of add() method.

**public** **boolean** add(E e) {

add(size(), e);//Inserts element at the end of list.

**return** **true**;

}

add(int index, E e)

**public** **void** add(**int** index, E element) {

**throw** **new** UnsupportedOperationException();

}

This implementation always thow UnsupportedOperationException(). The reason is that it is concrete class’s responsibility to implement this method.

get(int index)

Returns the specified element from the index in list. It throws IndexOutOfBoundsException.

**abstract** **public** E get(**int** index);

set(int index, E element)

This method is used to replace the specified element at given index in parameter. The element is replaced by the E element that is in parameter in set() method.

**public** E set(**int** index, E element) {

**throw** **new** UnsupportedOperationException();

}

add(int index, E element)

It inserts the element E at index as specified in parameter. It does not replace any element.

**public** **void** add(**int** index, E element) {

**throw** **new** UnsupportedOperationException();

}

remove(int index)

remove() method throws UnsupportedOperationException. But this method is not called directly. The concrete implementation of this method is provided by ArrayList<E>, LinkedList<E> class.

**public** E remove(**int** index) {

**throw** **new** UnsupportedOperationException();

}

indexOf(object o)

Returns the index of the Object o in List<>. Returns -1 if Object o is not found.

/\*\*

\* Returns index of the Object o passed as parameter.

\* This method uses listIterator() method to traverse in list.

\* \*/

**public** **int** indexOf(Object o) {

ListIterator<E> it = listIterator();

/\*\*

\* If the Object to be found is null then search for null

\* in List

\* \*/

**if** (o==**null**) {

**while** (it.hasNext())

/\*\*

\* If the next element is null then return the previous index

\* We return the previous index because we used next() method

\* in if condition.

\* \*/

**if** (it.next()==**null**)

**return** it.previousIndex();

}

/\*\*

\* Object o is not null.

\* traverse through List and check for object equality.

\* \*/

**else** {

//while elements exists in List.

**while** (it.hasNext())

/\*\*

\* If the next element equals the Object o then return the previous index

\* We return the previous index because we used next() method

\* in if condition.

\* \*/

**if** (o.equals(it.next()))

**return** it.previousIndex();

}

/\*\*

\* List is scanned and element is not found.

\* return -1.

\* \*/

**return** -1;

}

lastIndexOf(Object o)

This method returns the last index of the Object o. This method uses ListIterator that starts from end of the list.

/\*\*

\* Returns last index of the Object o passed as parameter.

\* This method uses listIterator(size()) method to traverse in list.

\* size() returns the total size of list.

\* This method calls the ListItr(int index) constructor.

\* \*/

**public** **int** lastIndexOf(Object o) {

ListIterator<E> it = listIterator(size());

/\*\*

\* If the Object to be found is null then search for null

\* in List

\* \*/

**if** (o==**null**) {

**while** (it.hasPrevious())

/\*\*

\* If the previous element is null then return the next index

\* We return the next index because we used previous() method

\* in if condition.

\* \*/

**if** (it.previous()==**null**)

**return** it.nextIndex();

}

/\*\*

\* Object o is not null.

\* traverse through List and check for object equality.

\* \*/

**else** {

//while elements exists in List.

**while** (it.hasPrevious())/\*\*

\* If the previous element equals the Object o then return the next index

\* We return the next index because we used previous() method

\* in if condition.

\* \*/

**if** (o.equals(it.previous()))

**return** it.nextIndex();

}

/\*\*

\* List is scanned and element is not found.

\* return -1.

\* \*/

**return** -1;

}

listIterator() and listIterator(int index)

It calls the overloaded version of ListIterator() method.

**public** ListIterator<E> listIterator() {

**return** listIterator(0);

}

listIterator(int index)

**public** ListIterator<E> listIterator(**final** **int** index) {

rangeCheckForAdd(index);

**return** **new** ListItr(index);

}

To understand [ListIterator<E>](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-9-listiterator.html) interface and its working in [ArrayList](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-5list-interface.html) follow this [link](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-9-listiterator.html).

subList(int fromIndex, int toIndex)

**public** List<E> subList(**int** fromIndex, **int** toIndex) {

**return** (**this** **instanceof** RandomAccess ?

**new** RandomAccessSubList<>(**this**, fromIndex, toIndex) :

**new** SubList<>(**this**, fromIndex, toIndex));

}

If the instance of this list is of RandomAccess i.e ArrayList then it returns RandomAccessSubList else it returns SubList.

RandomAccess is marker interface i.e, interface without any methods. It is used to indicate that it supports the fast random access.

equals(Object o)

Checks if 2 lists are same or not. More precisely both should be instance of List, both lists must have same elements & same size. Below is equals(Object o) method with comments.

/\*\*

\* This method is used to compare this list with the parameter

\* Object o for equality.

\* If both of them are list, both have same size, both have same element pairs

\* then only lists are same else return false.

\* \*/

**public** **boolean** equals(Object o) {

/\*\*

\* If o and this are same the return true

\* \*/

**if** (o == **this**)

**return** **true**;

/\*\*

\* If o is not instance of List then return false.

\* \*/

**if** (!(o **instanceof** List))

**return** **false**;

//Take listIterator() for this list.

ListIterator<E> e1 = listIterator();

/\*\*

\* Now o is instance of List so we will case it to List<?> and get

\* listIterator() for it.

\* \*/

ListIterator<?> e2 = ((List<?>) o).listIterator();

/\*\*

\* Now we compare if both of them has next or not.

\* Reason is if one list's size is less then another

\* than this test will fail.

\* Pretty neat approach

\* \*/

**while** (e1.hasNext() && e2.hasNext()) {

E o1 = e1.next();

Object o2 = e2.next();

//Comparaision of elements of both lists

**if** (!(o1==**null** ? o2==**null** : o1.equals(o2)))

**return** **false**;

}

/\*\*

\* If any of list has next element then lists are not same

\* return false.

\* \*/

**return** !(e1.hasNext() || e2.hasNext());

}

hashCode()

computes the hashCode for list.

**public** **int** hashCode() {

**int** hashCode = 1;

**for** (E e : **this**)

hashCode = 31\*hashCode + (e==**null** ? 0 : e.hashCode());

**return** hashCode;

}

removeRange(int fromIndex, int toIndex)

removes range of elements as specified in parameter.

**protected** **void** removeRange(**int** fromIndex, **int** toIndex) {

ListIterator<E> it = listIterator(fromIndex);

**for** (**int** i=0, n=toIndex-fromIndex; i<n; i++) {

it.next();

it.remove();

}

}

That’s all on AbstractList<E> class. Next is AbstractSequentialList<E>