CSCI 2120:
Software Design & Development II

UNIT 2: Collections Framework & Generics

ListIterator

Overview

1. Introduction
2. How to create a ListIterator object?
3. Methods of ListIterator
    a. Example 1: Iterate elements in forward direction.
    b. Example 2: Iterate bidirectional - forwards and backwards
    c. Example 3: Get index of the next and previous element
    d. Example 4: Use ListIterator to add and set elements into List
    e. Example 5: Use ListIterator to remove element from list
    f. Example 6: Use ListIterator without the generic types
4. Advantages of ListIterator
5. Limitations of ListIterator
6. Similarities between Iterator and ListIterator
7. Difference between Iterator and ListIterator

Introduction

ListIterator

`ListIterator` in Java is the most powerful iterator or cursor that was introduced in Java 1.2 version. It is a bi-directional cursor.

ListIterator Hierarchy

`ListIterator` is an interface *(an extension of `Iterator` interface)* that is used to retrieve the elements from a Collection object in both forward and reverse directions.

ListIterator API

It adds an additional six methods that reflect the bidirectional nature of a list iterator. By using `ListIterator`, we can perform different kinds of operations such as read, remove, replacement (current object), and the addition of the new elements. Java ListIterator can be used for all List implemented classes such as *`ArrayList`*, *`CopyOnWriteArrayList`*, *`LinkedList, Stack, Vector`*, etc.

How to create a ListIterator object?

listIterator()

To create a `ListIterator` object must call the `listIterator()` method of the `List` interface.

Syntax

The general syntax for creating ListIterator object in Java is as follows:

```
public ListIterator listIterator()  // return type is ListIterator.
```

Syntax for creating ListIterator object:

```
ListIterator<Type> litr = l.listIterator();
// l is any list object and Type is type of objects being iterated.
```

For example:

```
ListIterator<String> litr = l.listIterator(); // Get a full list iterator.

// Create a list iterator object that starts at index 3 in the forward direction.
ListIterator<String> litr = l.listIterator(3);
```

How to create a ListIterator object?

Key point:

When we use `ListIterator<String>`, the return type of `next()` method is `String`. In general, the return type of `next()` method matches the ListIterator's type parameter *(which indicates the type of the elements in the list)*.

Methods of ListIterator

Since `ListIterator` interface extends the `Iterator` interface to add a bidirectional traversal of the list. Therefore, all the methods provided by `Iterator` are available by default to the `ListIterator`. `ListIterator` interface has a total of 9 methods.

They are three type of methods:

- Forward traversal methods
- Backward traversal methods
- Mutate the collection methods

Methods of ListIterator

Forward direction:

1. **`public boolean hasNext()`:**
   This method returns true if the ListIterator has more elements when iterating the list in the forward direction.

2. **`public Object next()`:**
   This method returns the next element in the list. The return type of next() method is Object.

3. **`public int nextIndex()`:**
   This method returns the index of the next element in the list. The return type of this method is an integer.

Methods of ListIterator

Backward direction:

4. `public boolean hasPrevious()`:

   It checks that list has more elements in the backward direction. If the list has more elements, it will return true. The return type is boolean.

5. `public Object previous()`:

   It returns the previous element in the list and moves the cursor position backward direction. The return type is Object.

6. `public int previousIndex()`:

   It returns the index of the previous element in the list. The return type is an Integer.

Methods of ListIterator

Other capability methods:

7. **`public void remove():`**
   This method removes the last element returned by next() or previous() from the list. The return type is ‘nothing’.

8. **`public void set(Object o)`:**
   This method replaces the last element returned by next() or previous() with the new element.

9. **`public void add(Object o)`:**
   This method is used to insert a new element in the list.

Methods of ListIterator

Example Code

Let's implement all of these methods in the following example programs given in the next sections.

- Example 1: Iterate elements in forward direction.
- Example 2: Iterate bidirectional - forwards and backwards
- Example 3: Get index of the next and previous element
-

Example 1: Iterate elements in forward direction.

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorTest1 {
   public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A"); // Adding element A at index 0.
        list.add("B"); // Adding element B at index 1.
        list.add("C"); // Adding element C at index 2.

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

        // Create the list iterator object by calling listIterator() method. | in the comments indicates the position of iterator.
        ListIterator<String> iterator = list.listIterator();                    // |ABC
        System.out.println("List Iterator in Forward Direction:");

        // Call hasNext() method to check elements are present in forward direction.
        boolean elementsPresent = iterator.hasNext();                           // Return true.
        System.out.println(elementsPresent);

        int indexA = iterator.nextIndex();
        String elementA = iterator.next();                                      // A|BC
        System.out.println("IndexA = " +indexA +" "+ "Element: " +elementA);

        int indexB = iterator.nextIndex();
        String elementB = iterator.next();                                      // AB|C
        System.out.println("IndexB = " +indexB +" "+ "Element: " +elementB);

        int indexC = iterator.nextIndex();
        String elementC = iterator.next();                                      // ABC|
        System.out.println("IndexC = " +indexC +" "+ "Element: " +elementC);

        boolean elementsPresent2 = iterator.hasNext();                          // Return false because the iterator is at the end of the collection.
        System.out.println(elementsPresent2);
        String element = iterator.next();                                       // Throws NoSuchElementException because there is not next element.
   }
}
```

Example 1: Iterate elements in forward direction.

Output:

```
List: [A, B, C]
List Iterator in Forward Direction:
true
IndexA = 0 Element: A
IndexB = 1 Element: B
IndexC = 2 Element: C
false
Exception in thread "main" java.util.NoSuchElementException Create breakpoint
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:970)
    at ListIteratorTest1.main(ListIteratorTest1.java:37)
```

Example 2: Iterate bidirectional - forwards and backwards

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorTest2 {
   public static void main(String[] args) {
       List<String> list = new LinkedList<>();
       list.add("A");
       list.add("B");
       list.add("C");

       // Creating ListIterator object.
       ListIterator<String> listIterator = list.listIterator();

       // Traversing elements in forwarding direction.
       System.out.println("Forward Direction Iteration:");

       while(listIterator.hasNext()) {
           System.out.println(listIterator.next());
       }
       // Traversing elements in the backward direction. The ListIterator cursor is at just after the last element.
       System.out.println("Backward Direction Iteration:");

       while(listIterator.hasPrevious()) {
           System.out.println(listIterator.previous());
       }
   }
}
```

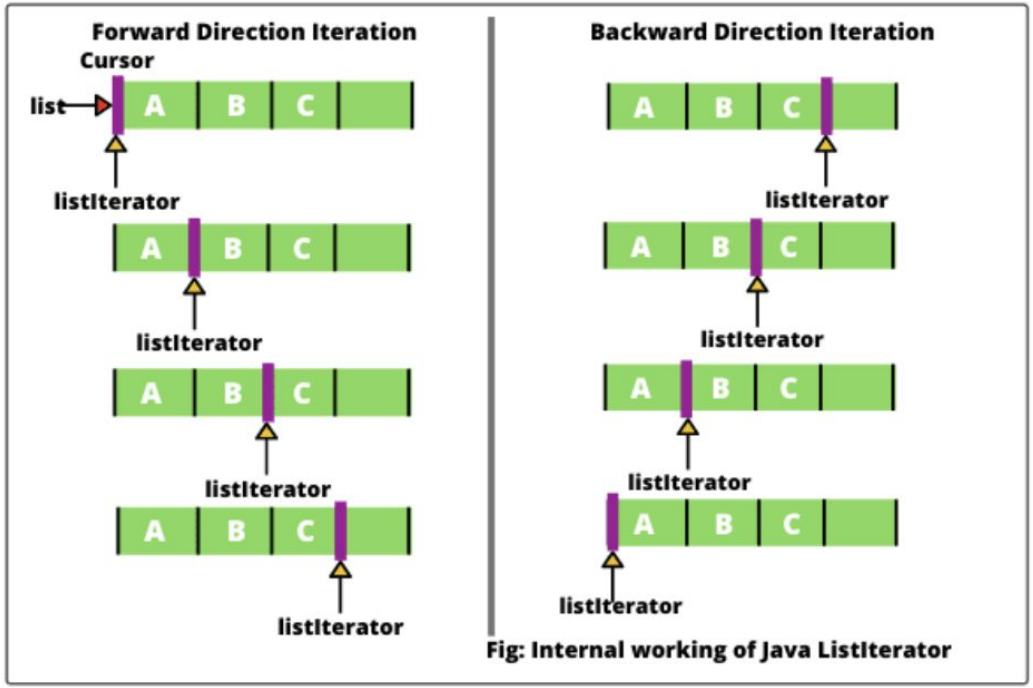
Example 2: Iterate bidirectional - forwards and backwards

Output:

```
Forward Direction Iteration:
A
B
C
Backward Direction Iteration:
C
B
A
```

Example 2: Iterate bidirectional - forwards and backwards

This is a diagram to show how `ListIterator` internally works in a backward direction.



1. ListIterator works in the backward direction the same as in the forward direction. When ListIterator's cursor reached right after to the last element in the list as shown in the figure, the call to hasPrevious() method checks that elements are present in the backward direction in the list. Since the elements are present in the backward direction in the list, so it will return true.

2. When previous() method is called, it returns the element and sets the position of the cursor for the next element in the backward direction. Look at the figure.

3. The call to hasPrevious() and previous() methods continue operations until ListIterator's cursor reaches the last element.

4. As soon as ListIterator's cursor points to the before the first element of the LinkedList, the hasPrevious() method returns a false value.

Example 2: Iterate bidirectional - forwards and backwards

Bidirectional Cursor

After observing the figure, we can see that the `hasPrevious()` and `previous()` methods do the same task but in the opposite direction of the `hasNext()` and `next()` methods.

Thus, we can say that `ListIterator` iterates elements of the list in both forward as well as backward directions. Therefore, it is also known as **bi-directional cursor**.

Example 3: Get index of the next and previous element

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorTest3 {
   public static void main(String[] args) {
       List<String> list = new ArrayList<>();
       list.add("Red");
       list.add("Green");
       list.add("Yellow");
       list.add("Orange");
       list.add("Blue");
       list.add("White");
       System.out.println("List: " + list);

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

       System.out.println();

       System.out.println("List Iterator in Forward Direction:");
       while (iterator.hasNext()) {
           int index = iterator.nextIndex();
           String element = iterator.next();
           System.out.println("Index = " + index + ", Element = " + element);
       }
       System.out.println();

       System.out.println("List Iterator in Backward Direction:");

       // Reuse the Java list iterator to iterate from the end to the beginning.
       while (iterator.hasPrevious()) {
           int index = iterator.previousIndex();
           String element = iterator.previous();
           System.out.println("Index = " + index + ", Element = " + element);
       }
   }
}
```

Example 3: Get index of the next and previous element

Output:

```
List: [Red, Green, Yellow, Orange, Blue, White]

List Iterator in Forward Direction:
Index = 0, Element = Red
Index = 1, Element = Green
Index = 2, Element = Yellow
Index = 3, Element = Orange
Index = 4, Element = Blue
Index = 5, Element = White

List Iterator in Backward Direction:
Index = 5, Element = White
Index = 4, Element = Blue
Index = 3, Element = Orange
Index = 2, Element = Yellow
Index = 1, Element = Green
Index = 0, Element = Red
```

Key Points:

1. If we use the next() method followed by the previous() method, the list iterator goes back to the same position.
2. The call to the next() method moves one index forward.
3. The call to the previous() method moves it one index backward.

Example 4: Use ListIterator to add and set elements into List

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class ListIteratorTest4 {
   public static void main(String[] args) {
       List<String> list = new ArrayList<>();                    // Create an object of ArrayList of String type.

       // Adding elements to arraylist.
       list.add("A");
       list.add("B");
       list.add("C");
       list.add("D");
       System.out.println("List: "+list);

       System.out.println();

       ListIterator<String> listIterator = list.listIterator();
       System.out.println("Forward Direction Iteration:");
       while(listIterator.hasNext()) {
           System.out.println(listIterator.next());
       }
       listIterator.add("E");                                    // Adds an element before the iterator position.
       System.out.println();
       System.out.println(list);

       System.out.println();

       System.out.println("Backward Direction Iteration:");
       while(listIterator.hasPrevious()){
           System.out.println(listIterator.previous());
       }
       listIterator.set("J");                                    // It will update the last element returned by previous.
       System.out.println();
       System.out.println(list);
   }
}
```

Example 4: Use ListIterator to add and set elements into List

Output:

```
List: [A, B, C, D]

Forward Direction Iteration:
A
B
C
D

[A, B, C, D, E]

Backward Direction Iteration:
E
D
C
B
A

[J, B, C, D, E]
```

Example 5: Use ListIterator to remove element from list

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorTest5 {
   public static void main(String[] args) {
       List<String> list = new ArrayList<>();
       list.add("A");
       list.add("B");
       list.add("C");
       list.add("D");

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

       ListIterator<String> listIterator = list.listIterator();
       System.out.println("Forward Direction Iteration:");

       while(listIterator.hasNext()){
           System.out.println(listIterator.next());
       }
       listIterator.remove();                          // Removes the last element returned by next method.
       System.out.println("New List: " +list);
   }
}
```

Example 5: Use ListIterator to remove element from list

Output:

```
List: [A, B, C, D]
Forward Direction Iteration:
A
B
C
D
New List: [A, B, C]
```

Warning:

You need to be careful when calling the `remove()` method. It can be called only once after calling the `next()` or `previous()` method. If you call it immediately after a call to `add()` method, it throws an `IllegalStateException`.

Example 6: Use ListIterator without the generic types

Generics vs Type Casting

Let's do one more important example program where we will not use generic type. In this case, we will need to do type casting. We will also perform remove and add operations in this program. Look at the source code to understand better.

Note: This is for demo purposes only, you should always use Generics notation!

Example 6: Use ListIterator without the generic types

```
import java.util.ArrayList;
import java.util.ListIterator;
public class ListIteratorTest6 {
   public static void main(String[] args) {
       ArrayList al = new ArrayList();
       al.add("Apple");
       al.add("Orange");
       al.add("Banana");
       al.add("Guava");
       al.add("Pineapple");
       System.out.println(al);

       // Create the object of ListIterator using listIterator() method.
       ListIterator litr = al.listIterator();
       while(litr.hasNext()) {
           Object o = litr.next();
           String str = (String)o;                    // Type casting.
           if(str.equals("Orange")) {
               litr.remove();                         // It will remove orange from the list.
               System.out.println(al);
           }
           else if(str.equals("Guava")) {
               litr.add("Grapes");                    // Adding Grapes after guava.
               System.out.println(al);
           }
           else if(str.equals("Pineapple")) {
               litr.set("Pears");                     // Replacing Pineapple element.
               System.out.println(al);
           }
       }
   }
}
```

Example 6: Use ListIterator without the generic types

Output:

```
[Apple, Orange, Banana, Guava, Pineapple]
[Apple, Banana, Guava, Pineapple]
[Apple, Banana, Guava, Grapes, Pineapple]
[Apple, Banana, Guava, Grapes, Pears]
```

Advantages of ListIterator

ListIterator has several advantages:

1. ListIterator supports many operations such as read, remove, replacement, and the adding of new objects.
2. Using the List Iterator, we can perform Iteration in both forward and backward directions.
3. Methods of ListIterator are easy and simple to use.

Limitations of ListIterator

ListIterator is the most powerful cursor but it still has some limitations:

1. ListIterator is applicable only for List implemented objects. Therefore, it is not a universal Java cursor.
2. Thus, It is not applicable to whole collection API.

Similarities between Iterator and ListIterator

There are several similarities between Iterator and ListIterator cursors. They are as:

1. Both are introduced in Java 1.2 version.
2. Both are Iterators that are used to iterate elements of a collection object.
3. Both support read and delete operations.
4. Both support forward direction iteration.
5. Both are not legacy interfaces.

Difference between Iterator and ListIterator

Now we will see the main differences between Iterator vs ListIterator in Java. They are as:

Iterator	ListIterator
1. Java Iterator is applicable to the whole Collection API.	1. Java ListIterator is only applicable for List implemented classes such as ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector, etc.
2. It is a Universal Iterator.	2. It is not a Universal Iterator in Java.
3. Iterator supports only forward direction Iteration.	3. ListIterator supports both forward and backward direction iterations.
4. It is known as a uni-directional iterator.	4. It is also known as bi-directional iterator.
5. Iterator supports only read and delete operations.	5. ListIterator supports all the operations such as read, remove, replacement, and the addition of the new elements.
6. We can get the Iterator object by calling iterator() method.	6. We can create ListIterator object by calling listIterator() method.

END