CSCI 2120:
Software Design & Development II

UNIT 2: Collections Framework & Generics
Iterator in Java

Overview

1. Introduction
2. Iterable Interface
3. How to create an Iterator object in Java?
4. How Iterator works internally?
    a. Example 1: Create and Use an Iterator on a Collection
5. Iterator Methods
    a. Example 2: Iterate elements and remove the odds
    b. Example 3: Iterate all elements in ArrayList using iterator()
6. Advantage of Iterator
7. Limitation of Iterator

Introduction

Iterator

An **iterator in Java** is a special type of object that provides sequential (one-by-one) access to the elements of a Collection object. Iterators were introduced in Java 1.2 as part of the Collections Framework. It can be applied to any Collection object. By using Iterator, we can perform both read and remove operations. Iterator is the **universal Iterator** or cursor.

Iterator API

An Iterator object implements Iterator interface which is present in java.util.Iterator package. Therefore, to use an Iterator, you must import either java.util.Iterator or java.util.*.

Iterable Interface

Iterable interface

The `Collection` interface extends `Iterable` interface which is at the top of the Collection hierarchy. The `iterable` interface is in java.lang.Iterable package. It provides a uniform way to retrieve the elements one-by-one from a Collection object.

It provides just one method named iterator() which returns an instance of Iterator and provides access one-by-one to the elements in the Collection object.

Syntax

The general syntax of iterator() method is as follows:

```
iterator() : Iterator //Return type is Iterator.
```

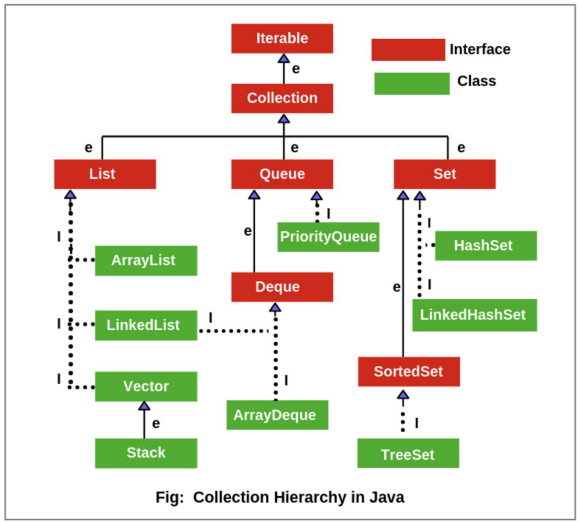


Fig: Collection Hierarchy in Java

How to create an Iterator object in Java?

Iterator object is created by calling iterator() method which is present in the iterable interface. The general syntax for creating Iterator object is as follows:

```
a) Iterator itr = c.iterator();                 // c is any collection object.
b) Iterator<Type> itr = c.iterator();           // Generic type.

For example:
     Iterator<String> itr = c.iterator();
     Iterator<Integer> itr = c.iterator();
```

How Iterator works internally?

The basic working mechanism for using Java Iterator is shown in the flow diagram.

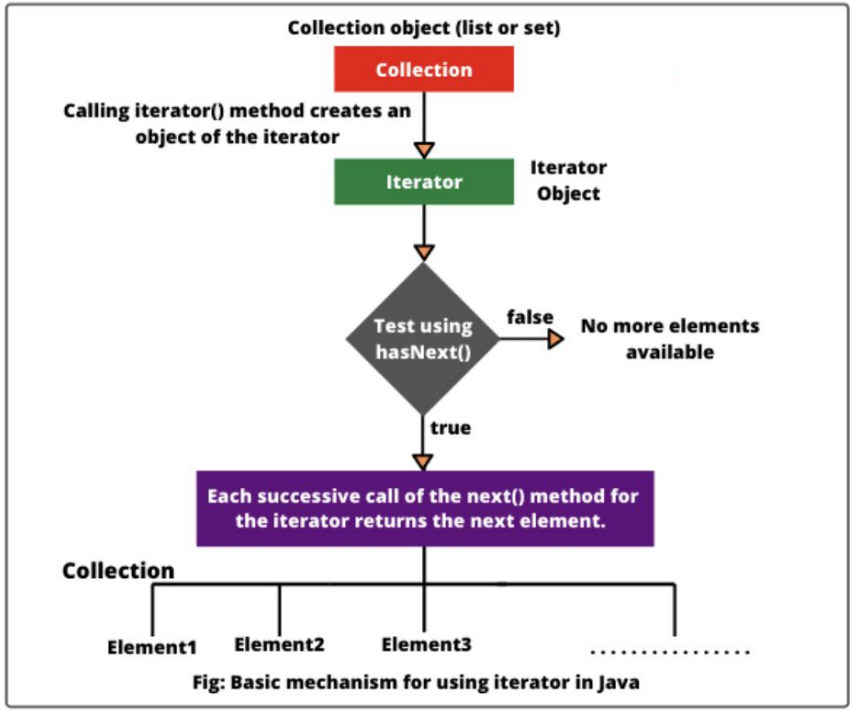


Fig: Basic mechanism for using iterator in Java

Example 1: Create and Use an Iterator on a Collection

```
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorTest1 {

   public static void main(String[] args) {

       ArrayList<String> al = new ArrayList<>();
       // Adding elements in the array list.
       al.add("A");
       al.add("B");
       al.add("C");
       al.add("D");
       al.add("E");
       al.add("F");

       Iterator<String> itr = al.iterator();
       while (itr.hasNext()) {
           String str = itr.next();
           System.out.print(str + " ");
       }
   }
}
```

Example 1: Create and Use an Iterator on a Collection

Output:

```
A B C D E F
```

When we create an iterator for the collection elements, the iterator looks like the following.

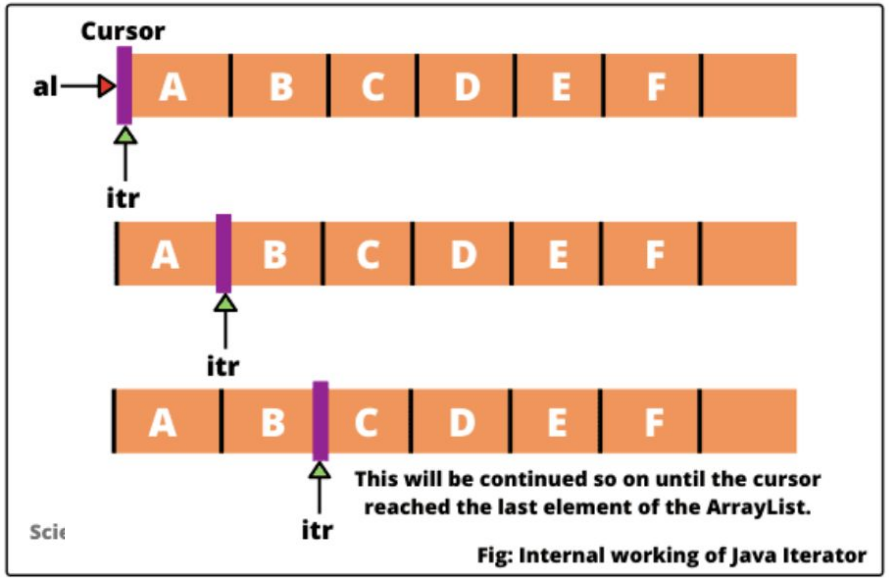


Fig: Internal working of Java Iterator

1. In the beginning, an Iterator points at right before the first element of the collection as shown in the above figure.

2. When hasNext() method is called on this Iterator, it returns true because elements are present in the forward direction.

With the call to the next() method, it returns an element from the collection and sets the Iterator object to the next element on the next call of this method.

3. Again hasNext() method will be called to check that elements are present for iteration. Since elements are present for iteration in the forward direction, therefore, it will return true. On call of next() method, it will return the next element from the collection and will set the Iterator object to the next element on the next call of this method.

4. Calling of hasNext() and next() method will be continued until the pointer moves to the last element.

5. When the pointer reached the last element of the ArrayList then the call to the hasNext() method will return false and the call to next() method will give an exception named NullPointerException.

Iterator Methods in Java

The Iterator interface provides three methods in Java. They are as follow:

1. **public boolean hasNext():** This method will return true if the iteration has more elements to traverse (iterate) in the forward direction. It will give false if all the elements have been iterated.

2. **public Object next():** The next() method return next element in the collection. It will throw NoSuchElementException when the iteration is complete.

3. **public void remove():** The remove() method removes the last or the most recent element returned by the iterator. It must be called after calling the next() method otherwise it will throw IllegalStateException.

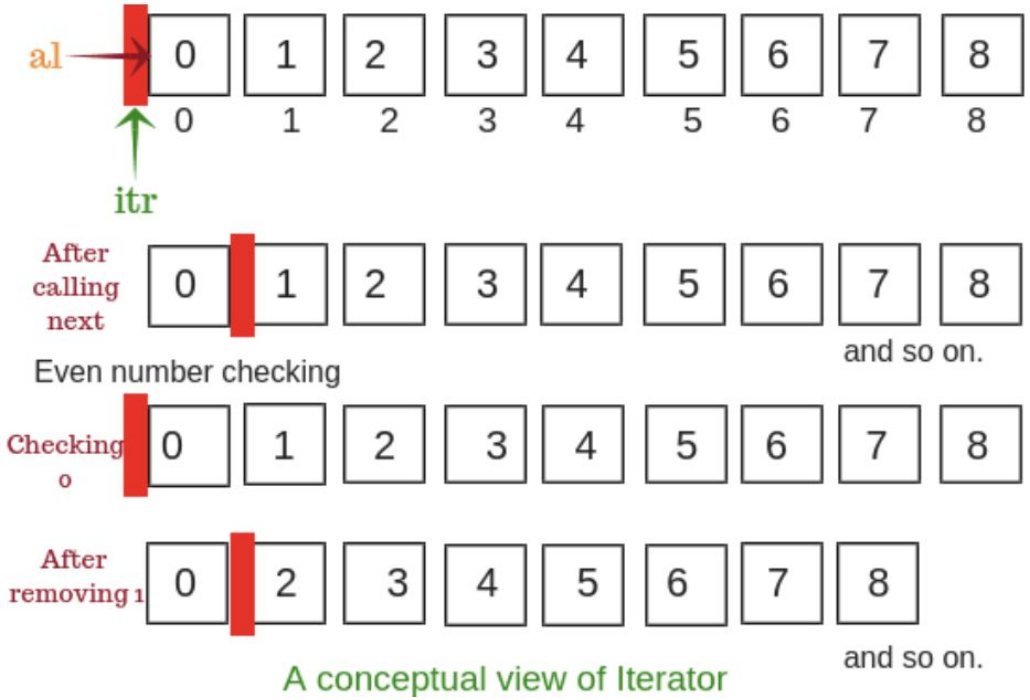
Let's take some example programs based on these methods in an easy way and step by step. We will use the Iterator concept to traverse all elements in the ArrayList.

Example 2: Iterate elements and remove odds

```
import java.util.ArrayList;
import java.util.Iterator;
public class IteratorTest2 {
   public static void main(String[] args) {
       // Create an object of ArrayList of type Integer.
       ArrayList<Integer> al = new ArrayList<Integer>();
       for (int i = 0; i <= 8; i++) {
           al.add(i);
       }
       System.out.println(al); // It will print all elements at a time.

       // Create the object of Iterator by calling iterator() method using reference variable al.
       // At the beginning, itr (cursor) will point to index just before the first element in al.
       Iterator<Integer> itr = al.iterator();

       // Checking the next element availability using reference variable itr.
       while (itr.hasNext()) {
           // Moving cursor to next element using reference variable itr.
           Integer i = itr.next(); // Here, Type casting does not require due to using of generic with Iterator.
           System.out.println(i);

           // Removing odd elements.
           if (i % 2 != 0) {
               itr.remove();
           }
           System.out.println(al);
       }
   }
}
```

Example 2: Iterate elements and remove odds

Look at the below picture to understand how elements are iterating in the above program.



Example 2: Iterate elements and remove odds

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
0
[0, 1, 2, 3, 4, 5, 6, 7, 8]
1
[0, 2, 3, 4, 5, 6, 7, 8]
2
[0, 2, 3, 4, 5, 6, 7, 8]
3
[0, 2, 4, 5, 6, 7, 8]
4
[0, 2, 4, 5, 6, 7, 8]
5
[0, 2, 4, 6, 7, 8]
6
[0, 2, 4, 6, 7, 8]
7
[0, 2, 4, 6, 8]
8
[0, 2, 4, 6, 8]
```

Example 3: Iterate all elements in ArrayList using iterator()

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class IteratorTest3 {
   public static void main(String[] args) {

       Collection<String> collection = new ArrayList<>();
       // Adding elements in the array list.
       collection.add("Red");
       collection.add("Green");
       collection.add("Black");
       collection.add("White");
       collection.add("Pink");

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

       while (iterator.hasNext()) {
           System.out.print(iterator.next().toUpperCase() + " ");
       }
       System.out.println();
   }
}
```

Example 3: Iterate all elements in ArrayList using iterator()

Output:

```
RED GREEN BLACK WHITE PINK
```

Advantage of Iterator

Java Iterator has the following advantages:

- An iterator can be used with any Collection classes.
- We can perform both read and remove operations.
- It acts as a universal cursor for Collection API.

Limitation of Iterator

Iterator has the following limitations or drawbacks:

- By using Iterator, we can move only towards the forward direction. We cannot move in the backward direction. Hence, these are called single-direction cursors.
- We can perform either read operation or remove operation.
- We cannot perform the replacement of new objects.
  - *Analogy: suppose there are five mangoes in a box. Out of five, two are spoiled but we cannot replace those damaged mangos with new mangos*.

To overcome the above drawbacks, we should use the ListIterator.

END