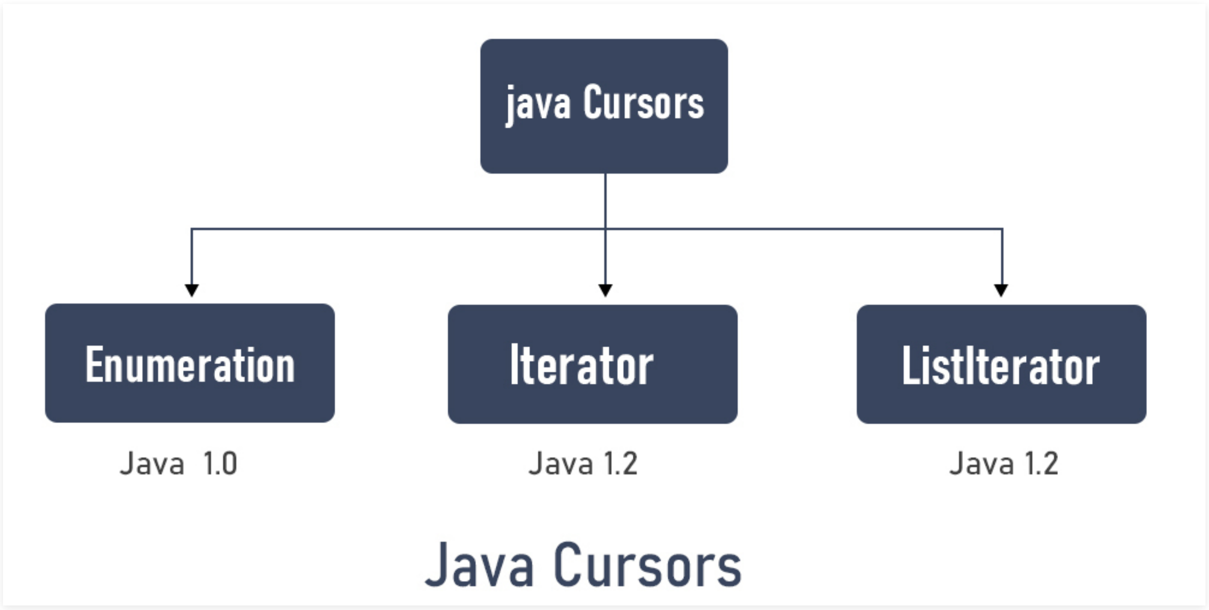
Iterators in Java

Difficulty Level : Medium Last Updated : 22 Dec, 2021

A Java Cursor is an Iterator, which is used to iterate or traverse or retrieve a Collection or Stream object's elements one by one. There are **three cursors in Java**.

1. Iterator
2. Enumeration
3. ListIterator

Note: *SplitIterator can also be considered as a cursor as it is a type of Iterator only.*



1. Iterator

Iterators in Java are used in the Collection framework to retrieve elements one by one. It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is an improved version of Enumeration with the additional functionality of removing an element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque, and all implemented

classes of Map interface. Iterator is the **only** cursor available for the entire collection framework.
Iterator object can be created by calling *iterator()* method present in Collection interface.

Syntax:

```
Iterator itr = c.iterator();
```

> ***Note:** Here "c" is any Collection object. itr is of type Iterator interface and refers to "c".*

Methods of Iterator Interface in Java

Iterator interface defines **three** methods as listed below:

1. hasNext(): Returns true if the iteration has more elements.

```
public boolean hasNext();
```

2. next(): Returns the next element in the iteration. It throws **NoSuchElementException** if no more element is present.

```
public Object next();
```

3. remove(): Removes the next element in the iteration. This method can be called only once per call to next().

```
public void remove();
```

> ***Note:** remove() method can throw two exceptions namely as follows:*
>
> - ***UnsupportedOperationException** : If the remove operation is not supported by this iterator*
> - ***IllegalStateException** : If the next method has not yet been called, or the remov e method has already been called after the last call to the next method.*

How does Java Iterator Works Internally?

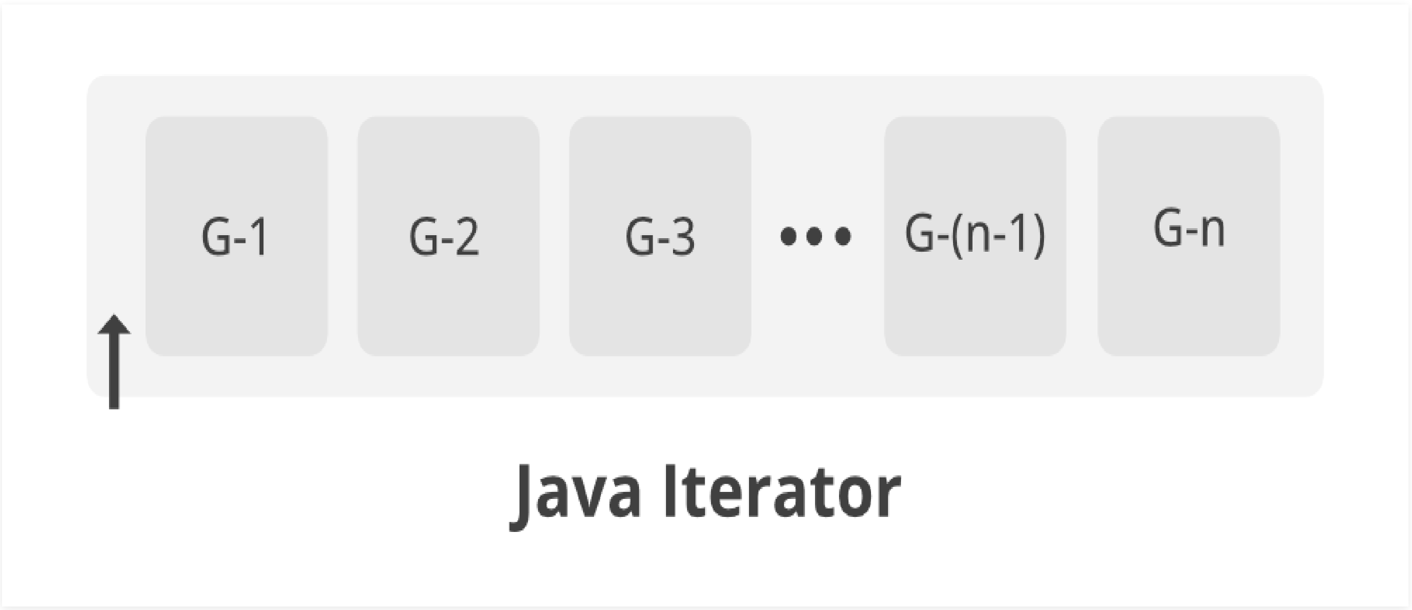
In this section, we will try to understand how Java Iterator and its methods work internally. Let us take the following LinkedList object to understand this functionality.

```
List<String> cities = new LinkedList<>();
cities.add("G-1");
cities.add("G-2");
cities.add("G-3");
.
.
.
cities.add("G-n");
```

Now, let us create an Iterator object on List object as shown below:

```
Iterator<String> citiesIterator = cities.iterator();
```

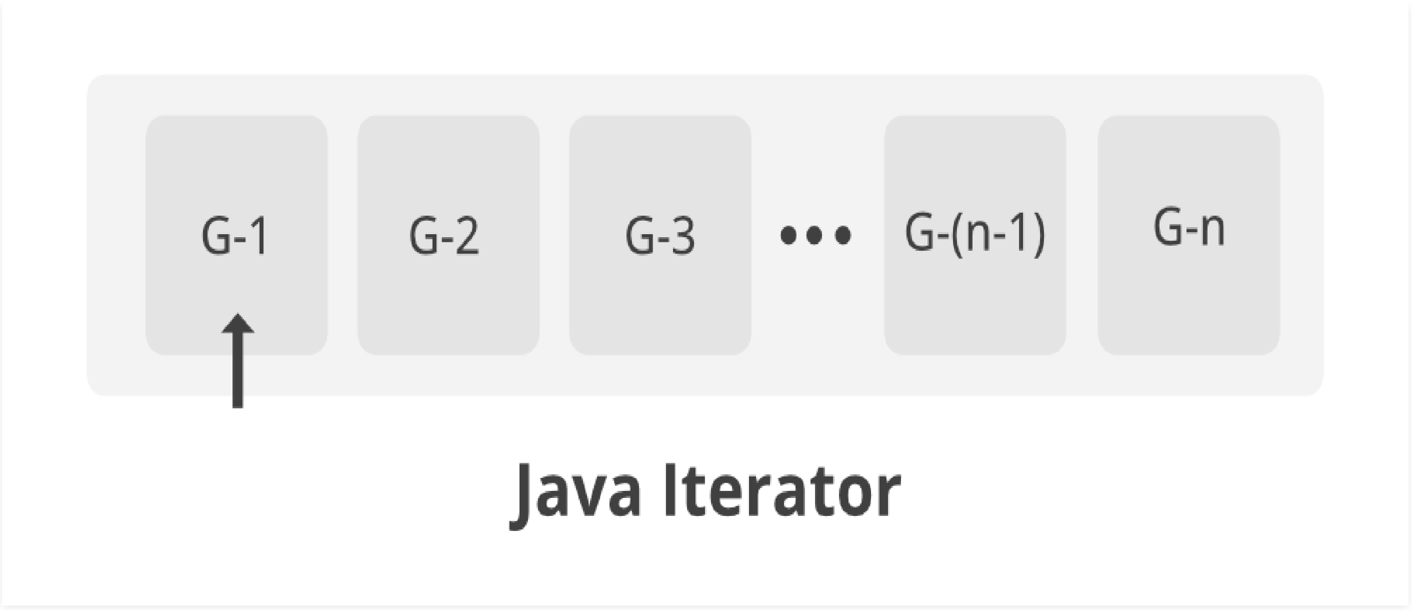
The “citiesIteartor” iterator will look like –



Here Iterator’s Cursor is pointing before the first element of the List.

Now, we will run the following code snippet.

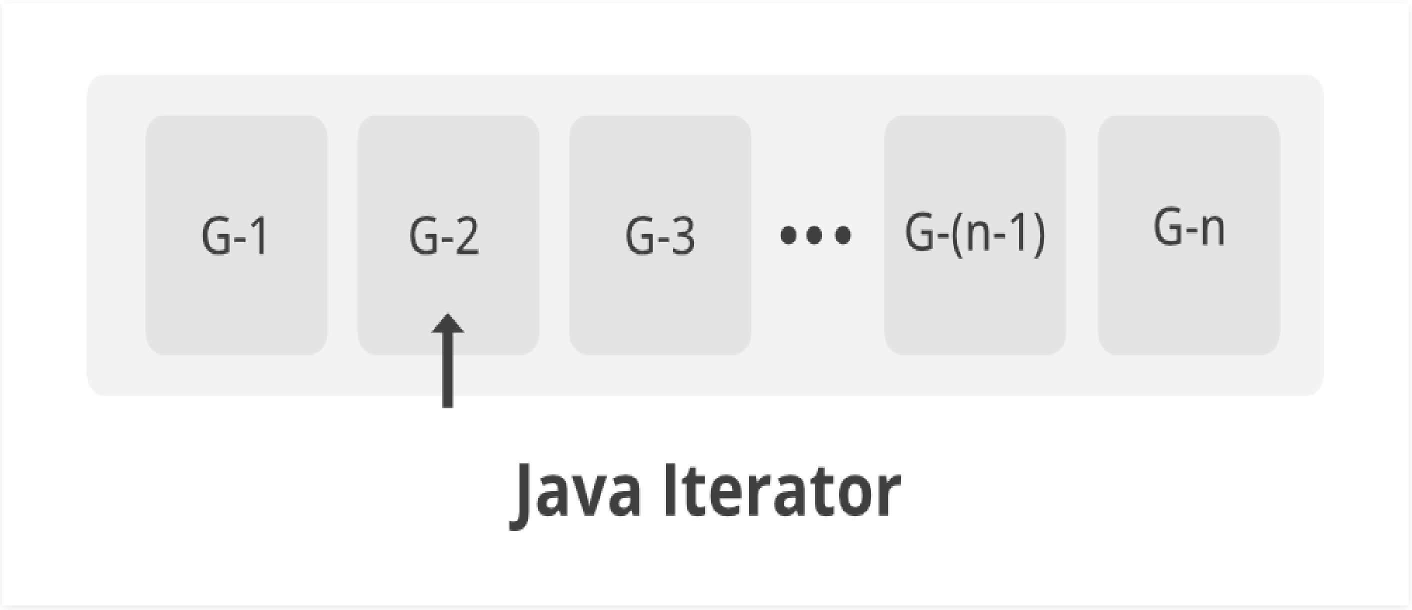
```
citiesIterator.hasNext();
citiesIterator.next();
```



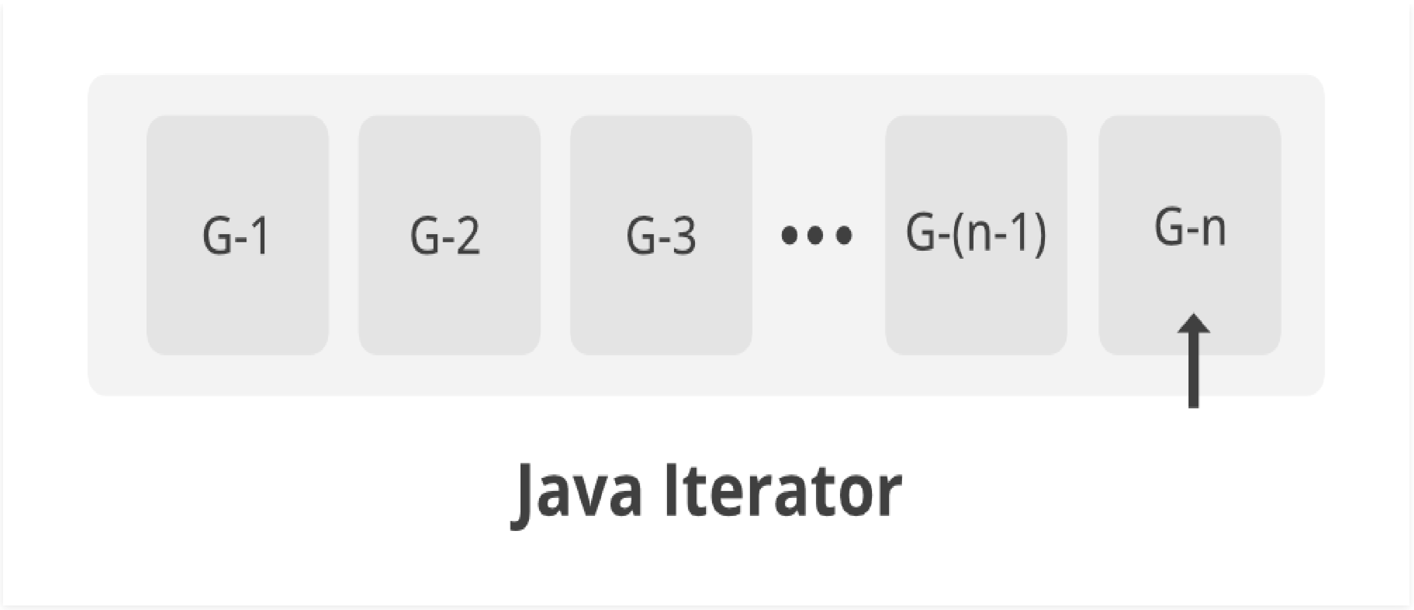
When we run the above code snippet, Iterator’s Cursor points to the first element in the list as shown in the above diagram.

Now, we will run the following code snippet.

```
citiesIterator.hasNext();
citiesIterator.next();
```

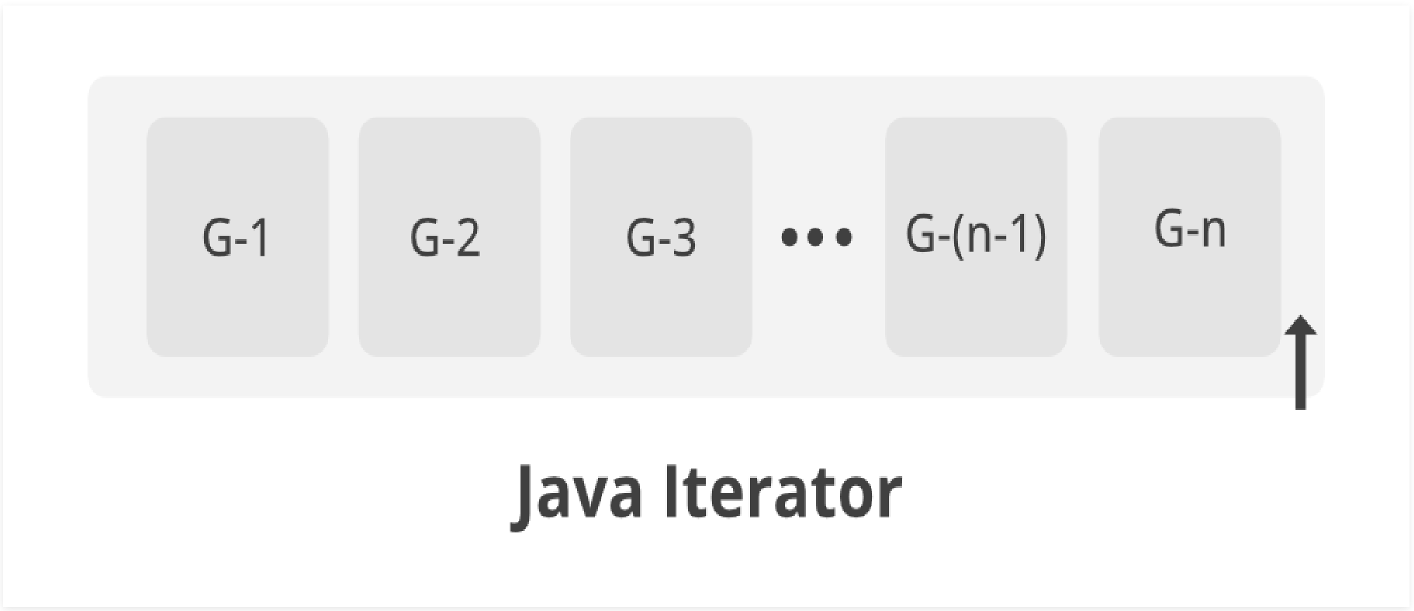


When we run the above code snippet, Iterator’s Cursor points to the second element in the list as shown in the above diagram. Do this process to reach the Iterator’s Cursor to the end element of the List.



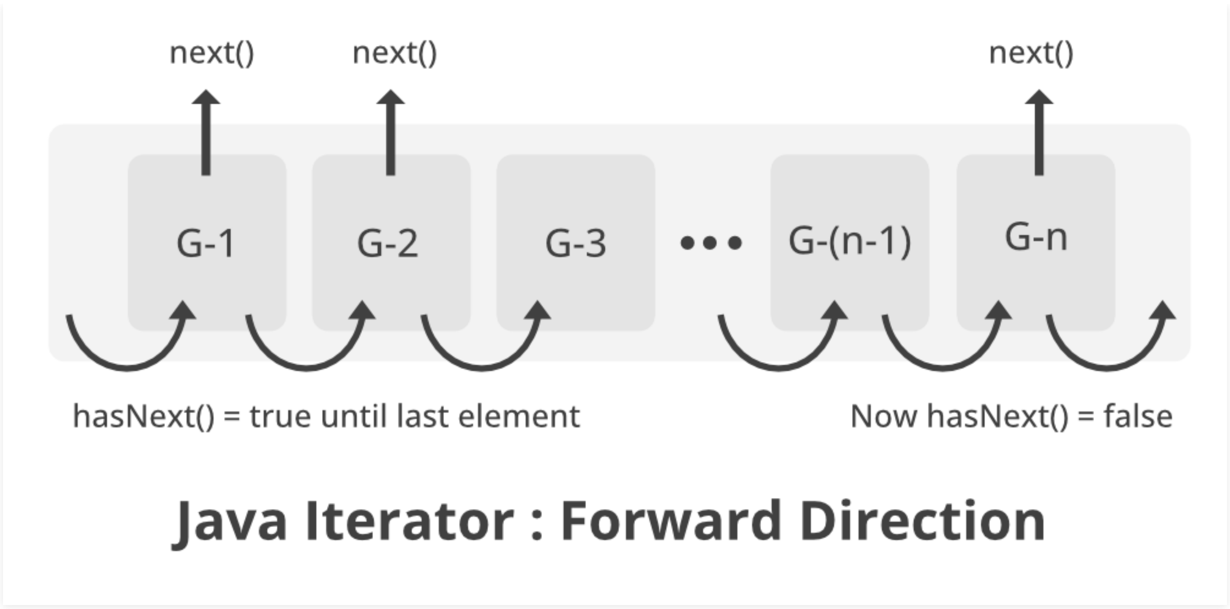
After reading the final element, if we run the below code snippet, it returns a “false” value.

```
citiesIterator.hasNext();
```



As Iterator’s Cursor points to the after the final element of the List, hasNext() method returns a false value.

> ***Note:*** *After observing all these diagrams, we can say that Java Iterator supports onl y Forward Direction Iteration as shown in the below diagram. So it is also known as Uni-Directional Cursor.*


next()
next()
next()
G-1
G-2
G-3
G-(n-1)
G-n
hasNext() = true until last element
Now hasNext() = false
Java Iterator : Forward Direction

Example:

```
// Java program to Demonstrate Iterator

// Importing ArrayList and Iterator classes
// from java.util package
import java.util.ArrayList;
import java.util.Iterator;

// Main class
public class Test {
    // Main driver method
    public static void main(String[] args)
    {
        // Creating an ArrayList class object
        // Declaring object of integer type
        ArrayList<Integer> al = new ArrayList<Integer>();

        // Iterating over the List
        for (int i = 0; i < 10; i++)
            al.add(i);

        // Printing the elements in the List
        System.out.println(al);

        // At the beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator itr = al.iterator();

        // Checking the next element  where
        // condition holds true till there is single element
        // in the List using hasnext() method
        while (itr.hasNext()) {
            //  Moving cursor to next element
            int i = (Integer)itr.next();

            // Getting even elements one by one
            System.out.print(i + " ");

            // Removing odd elements
            if (i % 2 != 0)
                itr.remove();
        }

        // Command for next line
        System.out.println();

        // Printing the elements inside the object
        System.out.println(al);
    }
}
```

Output

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[0, 2, 4, 6, 8]
```

SplitIterator

Spliterators, like other Iterators, are for traversing the elements of a source. A source can be a Collection, an IO channel, or a generator function. It is included in JDK 8 for support of efficient parallel traversal(parallel programming) in addition to sequential traversal. Java Spliterator interface is an internal iterator that breaks the stream into smaller parts. These smaller parts can be processed in parallel.

> ***Note:*** *In real life programming, we may never need to use Spliterator directly. Under normal operations, it will behave exactly the same as Java Iterator.*

Advantages of Java Iterator

- We can use it for any Collection class.
- It supports both READ and REMOVE operations.
- It is a Universal Cursor for Collection API.
- Method names are simple and easy to use them.

Also, there are certain limitations of Iterator which are listed as follows:

Limitations of Java Iterator

- In CRUD Operations, it does NOT support CREATE and UPDATE operations.
- It supports only Forward direction iteration that is a Uni-Directional Iterator.
- Compare to Spliterator, it does NOT support iterating elements parallel which means it supports only Sequential iteration.
- Compare to Spliterator, it does NOT support better performance to iterate large volume of data.

2. Enumeration

It is an interface used to get elements of legacy collections(Vector, Hashtable). Enumeration is the first iterator present from JDK 1.0, rests are included in JDK 1.2 with more functionality. Enumerations are also used to specify the input streams to a *SequenceInputStream*. We can create an Enumeration object by calling *elements()* method of vector class on any vector object

```
// Here "v" is an Vector class object. e is of
// type Enumeration interface and refers to "v"
Enumeration e = v.elements();
```

There are **two** methods in the Enumeration interface namely :

1. public boolean hasMoreElements(): This method tests if this enumeration contains more elements or not.

2. public Object nextElement(): This method returns the next element of this enumeration. It throws NoSuchElementException if no more element is present

```
// Java program to demonstrate Enumeration

// Importing Enumeration and Vector classes
// from java.util package
import java.util.Enumeration;
import java.util.Vector;

// Main class
```

```
public class Test
{
    // Main driver method
    public static void main(String[] args)
    {
        // Creating a vector object
        Vector v = new Vector();

      // Iterating over vector object
      for (int i = 0; i < 10; i++)
            v.addElement(i);

      // Printing elements in vector object
      System.out.println(v);

        // At beginning e(cursor) will point to
        // index just before the first element in v
        Enumeration e = v.elements();

        // Checking the next element availability where
        // condition holds true till threre is a single element
      // remaining in the List
        while (e.hasMoreElements())
        {
            // Moving cursor to next element
            int i = (Integer)e.nextElement();

            // Print above elements in object
            System.out.print(i + " ");
        }
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
```

There are certain limitations of enumeration which are as follows:

- Enumeration is for **legacy** classes(Vector, Hashtable) only. Hence it is not a universal iterator.
- Remove operations can’t be performed using Enumeration.
- Only forward direction iterating is possible.

Similarities between Java Enumeration and Iterator

- Both are Java Cursors.
- Both are used to iterate a Collection of object elements one by one.
- Both support READ or Retrieval operation.
- Both are Uni-directional Java Cursors which means support only Forward Direction Iteration.

Differences between Java Enumeration and Iterator

The following table describes the differences between Java Enumeration and Iterator:

Enumeration	Iterator
Introduced in Java 1.0	Introduced in Java 1.2
Legacy Interface	Not Legacy Interface
It is used to iterate only Legacy Collection classes.	We can use it for any Collection class.
It supports only READ operation.	It supports both READ and DELETE operations.
It's not Universal Cursor.	It is a Universal Cursor.
Lengthy Method names.	Simple and easy-to-use method names.

3. ListIterator

It is only applicable for List collection implemented classes like ArrayList, LinkedList, etc. It provides bi-directional iteration. ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator. ListIterator object can be created by calling *listIterator()* method present in the List interface.

```
ListIterator ltr = l.listIterator();
```

> ***Note:*** *Here "l" is any List object, ltr is of type. ListIterator interface and refers to "l". L istIterator interface extends the Iterator interface. So all three methods of Iterator int*

*erface are available for ListIterator. In addition, there are **six** more methods.*

1. Forward direction

1.1 hasNext(): Returns true if the iteration has more elements

```
public boolean hasNext();
```

1.2 next(): Same as next() method of Iterator. Returns the next element in the iteration.

```
public Object next();
```

1.3 nextIndex(): Returns the next element index or list size if the list iterator is at the end of the list.

```
public int nextIndex();
```

2. Backward direction

2.1 hasPrevious(): Returns true if the iteration has more elements while traversing backward.

```
public boolean hasPrevious();
```

2.2 previous(): Returns the previous element in the iteration and can throw **NoSuchElementException** if no more element present.

```
public Object previous();
```

2.3 previousIndex(): Returns the previous element index or -1 if the list iterator is at the beginning of the list,

```
public int previousIndex();
```

3. Other Methods

3.1 remove(): Same as remove() method of Iterator. Removes the next element in the iteration.

```
public void remove();
```

3.2 set(Object obj): Replaces the last element returned by next() or previous() with the specified element.

```
public void set(Object obj);
```

3.3 add(Object obj): Inserts the specified element into the list at the position before the element that would be returned by next()

```
public void add(Object obj);
```

Clearly, the three methods that *ListIterator* inherits from Iterator (*hasNext()*, *next()*, and *remove()*) do exactly the same thing in both interfaces. The *hasPrevious()* and the previous operations are exact analogues of *hasNext()* and *next()*. The former operations refer to the element before the (implicit) cursor, whereas the latter refers to the element after the cursor. The previous operation moves the cursor backward, whereas the next moves it forward.

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().*

1. set() method can throw 4 exceptions.

- *UnsupportedOperationException:* if the set operation is not supported by this list iterator
- *ClassCastException:* If the class of the specified element prevents it from being added to this list
- *IllegalArgumentException:* If some aspect of the specified element prevents it from being added to this list
- *IllegalStateException:* If neither next nor previous have been called, or remove or add have been called after the last call to next or previous

2. add() method can throw 3 exceptions.

- *UnsupportedOperationException:* If the add method is not supported by this list iterator
- *ClassCastException:* If the class of the specified element prevents it from being added to this list
- *IllegalArgumentException:* If some aspect of this element prevents it from being added to this list

Example:

```java
// Java program to demonstrate ListIterator

// Importing ArrayList and List iterator classes
// from java.util package
import java.util.ArrayList;
import java.util.ListIterator;

// Main class
public class Test {
    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of ArrayList class
        ArrayList al = new ArrayList();

        // Iterating over Arraylist object
        for (int i = 0; i < 10; i++)

            // Adding elements to the Arraylist object
            al.add(i);

        // Print and display all elements inside object
        // created above
        System.out.println(al);

        // At beginning ltr(cursor) will point to
        // index just before the first element in al
        ListIterator ltr = al.listIterator();

        // Checking the next element availability
        while (ltr.hasNext()) {
            //  Moving cursor to next element
            int i = (Integer)ltr.next();

            // Getting even elements one by one
            System.out.print(i + " ");

            // Changing even numbers to odd and
            // adding modified number again in
            // iterator
            if (i % 2 == 0) {
                // Change to odd
                i++;
                // Set method to change value
                ltr.set(i);
                // To add
                ltr.add(i);
            }
        }

        // Print and display statements
        System.out.println();
        System.out.println(al);
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[1, 1, 1, 3, 3, 3, 5, 5, 5, 7, 7, 7, 9, 9, 9]
```

> ***Note:*** *Similarly, there are certain limitations with ListIterator. It is the most powerful i terator but it is only applicable for List implemented classes, so it is not a universal it erator.*

Important Points

1. Please note that initially, any iterator reference will point to the index just before the index of the first element in a collection.
2. We don't create objects of Enumeration, Iterator, ListIterator because they are interfaces. We use methods like elements(), iterator(), listIterator() to create objects. These methods have an anonymous Inner Class that extends respective interfaces and return this class object.

> ***Note:*** *The **$** symbol in reference class name is a proof that concept of inner classes is used and these class objects are created.*

This can be verified by the below code. For more on inner class refer

```
// Java program to demonstrate iterators references

// Importing required classes from java.util package
import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating an object of Vector class
        Vector v = new Vector();

        // Creating three iterators
        Enumeration e = v.elements();
        Iterator itr = v.iterator();
        ListIterator ltr = v.listIterator();

        // Print class names of iterators
        // using getClass() and getName() methods
        System.out.println(e.getClass().getName());
        System.out.println(itr.getClass().getName());
        System.out.println(ltr.getClass().getName());
    }
}
```

Output

```
java.util.Vector$1
java.util.Vector$Itr
java.util.Vector$ListItr
```