

Collections Contd...

We have seen several frameworks like ArrayList, LinkedList, ArrayDeque, PriorityQueue, TreeSet, HashSet, LinkedHashSet let us see how to access each one of these

ArrayList and LinkedList

It is clear from previous sessions that values in both these framework have **indexes**, so indexes can be accessed using **for loop**. But let's be sure about it by actually seeing the output

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        ArrayList x = new ArrayList();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(int i=0;i<=x.size()-1;i++)
        {
            System.out.println(x.get(i)); //individual value is accessed.
        }
    }
}
```



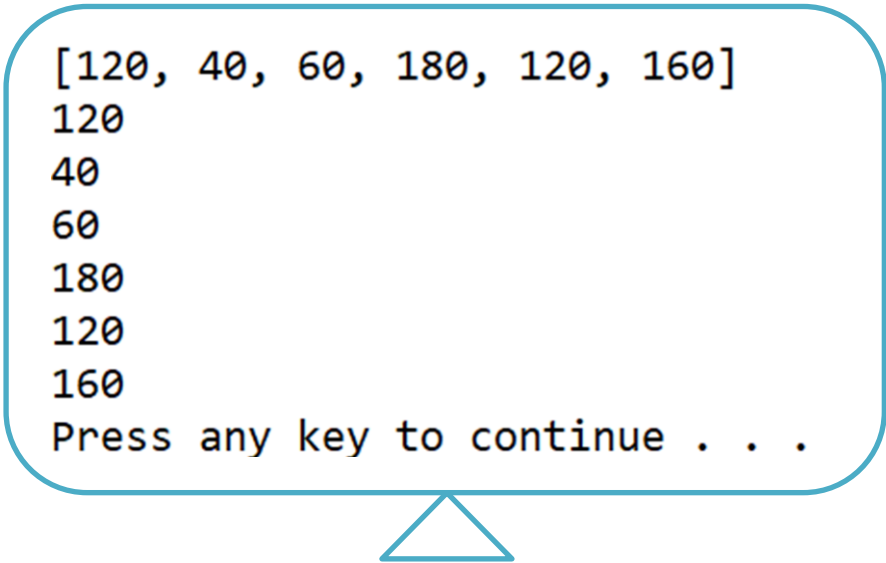
Output:

```
[120, 40, 60, 180, 120, 160]
120
40
60
180
120
160
Press any key to continue . . .
```

Now let's see for LinkedList

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        LinkedList x = new LinkedList();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(int i=0;i<=x.size()-1;i++)
        {
            System.out.println(x.get(i)); //individual value is accessed.
        }
    }
}
```

Output:



```
[120, 40, 60, 180, 120, 160]
120
40
60
180
120
160
Press any key to continue . . .
```

But that's not the case with other frameworks as they don't have indexes assigned to the values stored in them. So let's see how to access them

For each loop

The Java for-each loop or enhanced for loop is introduced since J2SE 5.0. It **provides an alternative approach to traverse the array or collection** in Java. It

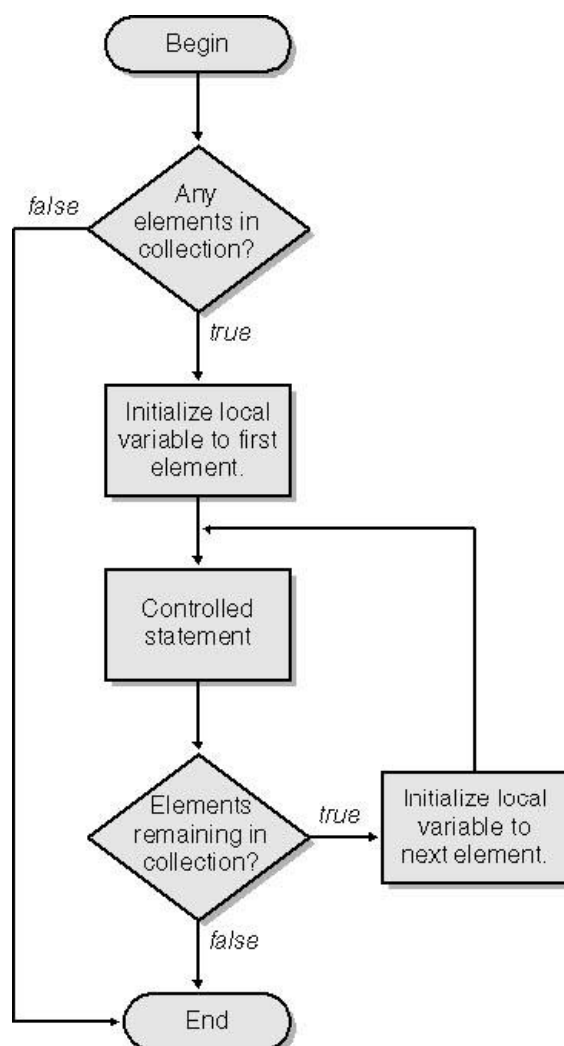
is mainly used to traverse the array or collection elements. The **advantage** of the **for-each loop** is that it **eliminates the possibility of bugs and makes the code more readable**. It is known as the for-each loop because it traverses each element one by one.

The **drawback** of the enhanced for loop is that it **cannot traverse the elements in reverse order**.

Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only.

But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Flowchart for foreach loop in java



Syntax:

```
for(data_type variable : array | collection)
{
    //body of for-each loop
}
```

Note: foreach loop can we used for array as well as collections.

How does it work?

The Java for-each loop traverses the array or collection until the last element. For each element, it stores the element in the variable and executes the body of the for-each loop.



Let's now see for how many frameworks does this loop works. We will take one example from each List, Queue, Set.

ArrayList from Lists

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        ArrayList x = new ArrayList();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(Object temp:x)
        {
            System.out.println(temp); //individual value is accessed.
        }
    }
}
```



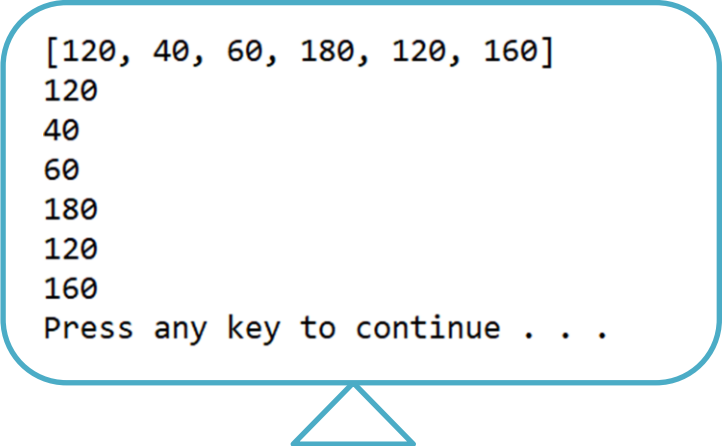
ArrayDeque from Queue

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        ArrayDeque x = new ArrayDeque();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(Object temp:x)
        {
            System.out.println(temp); //individual value is accessed.
        }
    }
}
```

TreeSet from Set

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        TreeSet x = new TreeSet();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        for(Object temp:x)
        {
            System.out.println(temp); //individual value is accessed.
        }
    }
}
```

Output:



```
[120, 40, 60, 180, 120, 160]
120
40
60
180
120
160
Press any key to continue . . .
```

Let us now see the next method to access the values.

Iterator

In Java, Iterator is an interface available in Collection framework in java.util package. It is a Java Cursor used to iterate a collection of objects.

Key Points

- It is used to traverse a collection object elements one by one.
- It is available since Java 1.2 Collection Framework.
- It is applicable for all Collection classes. So it is also known as Universal Java Cursor.
- It supports both READ and REMOVE Operations.
- Compare to Enumeration interface, Iterator method names are simple and easy to use.

Let us see one example and understand in better way

```
import java.util.*; //imports all the classes
class Demo
{
    public static void main(String[] args)
    {
        TreeSet x = new TreeSet();
        x.add(120);
        x.add(40);
        x.add(60);
        x.add(180);
        x.add(120);
        x.add(160);
        System.out.println(x);
        Iterator itr = x.iterator();
        while(itr.hasNext()==true)
        {
            System.out.println(itr.next());
        }
    }
}
```

Output:

```
[40, 60, 120, 160, 180]
40
60
120
160
180
Press any key to continue . . .
```

Iterator vs ListIterator:



1) **Iterator** is used for traversing **List and Set both**.

We can use **ListIterator** to traverse **List only**, we cannot traverse Set using ListIterator.

2) We can **traverse in only forward direction** using Iterator.

Using **ListIterator**, we can **traverse a List in both the directions** (forward and Backward).

3) We **cannot** obtain indexes while using **Iterator**

We **can obtain** indexes at any point of time while traversing a list using **ListIterator**. The methods `nextIndex()` and `previousIndex()` are used for this purpose.

4) We **cannot** add element to collection while traversing it using Iterator, it throws **ConcurrentModificationException** when you try to do it.

We **can add** element at any point of time while traversing a list using **ListIterator**.

5) We **cannot** replace the existing element value when using **Iterator**.

By using **set(E e)** method of **ListIterator** we can replace the last element returned by **next()** or **previous()** methods.

6) Methods of Iterator:

- **hasNext()**
- **next()**
- **remove()**

Methods of ListIterator:

- **add(E e)**
- **hasNext()**
- **hasPrevious()**
- **next()**
- **nextIndex()**
- **previous()**
- **previousIndex()**
- **remove()**
- **set(E e)**

**GREAT
WORK!**

