Object - Oriented Programming 2

Week 10. Introduction to Collection framework, Collection Interfaces, Collection Classes, Iterator and ListIterator

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Agenda

1. Introduction to Collection framework,
2. Collection Interfaces,
3. Collection Classes,
4. Iterator and ListIterator

Introduction to Collection framework

Introduction to Collection framework

in java, **a collection** is an object that collects multiple elements into a single unit. It is used to store, fetch and manipulate data. For example, list is used to collect elements and referred by a list object

Java **collection framework** represents a hierarchy of set of interfaces and classes that are used to manipulate group of objects.

Collections framework was added to Java 1.2 version. Prior to Java 2, Java provided adhoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects.

Java collections framework is contained **in java.util package**. It provides many important classes and interfaces to collect and organize group of objects.

Components of Collection Framework

The collections framework consists of:

1. Collection interfaces such as sets, lists, and maps. These are used to collect different types of objects.
2. Collection classes such as ArrayList, HashSet etc that are implementations of collection interfaces.
3. Concurrent implementation classes that are designed for highly concurrent use.
4. Algorithms that provides static methods to perform useful functions on collections, such as sorting a list.

Advantage of Collection Framework

1. Reduces programming effort by providing built-in set of data structures and algorithms.
2. Increases performance by providing high-performance implementations of data structures and algorithms.
3. Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.
4. Increase Productivity
5. Reduce operational time
6. versatility to work with current collection as well

Important Interfaces of Collection API

Interface	Description
Collection	Enables you to work with groups of object; it is at the top of Collection hierarchy
Deque	Extends Queue to handle double ended queue.
List	Extends Collection to handle sequences list of object.
Queue	Extends Collection to handle special kind of list in which element are removed only from the head.
Set	Extends Collection to handle sets, which must contain unique element.
SortedSet	Extends Set to handle sorted set.

Java Collection Hierarchy

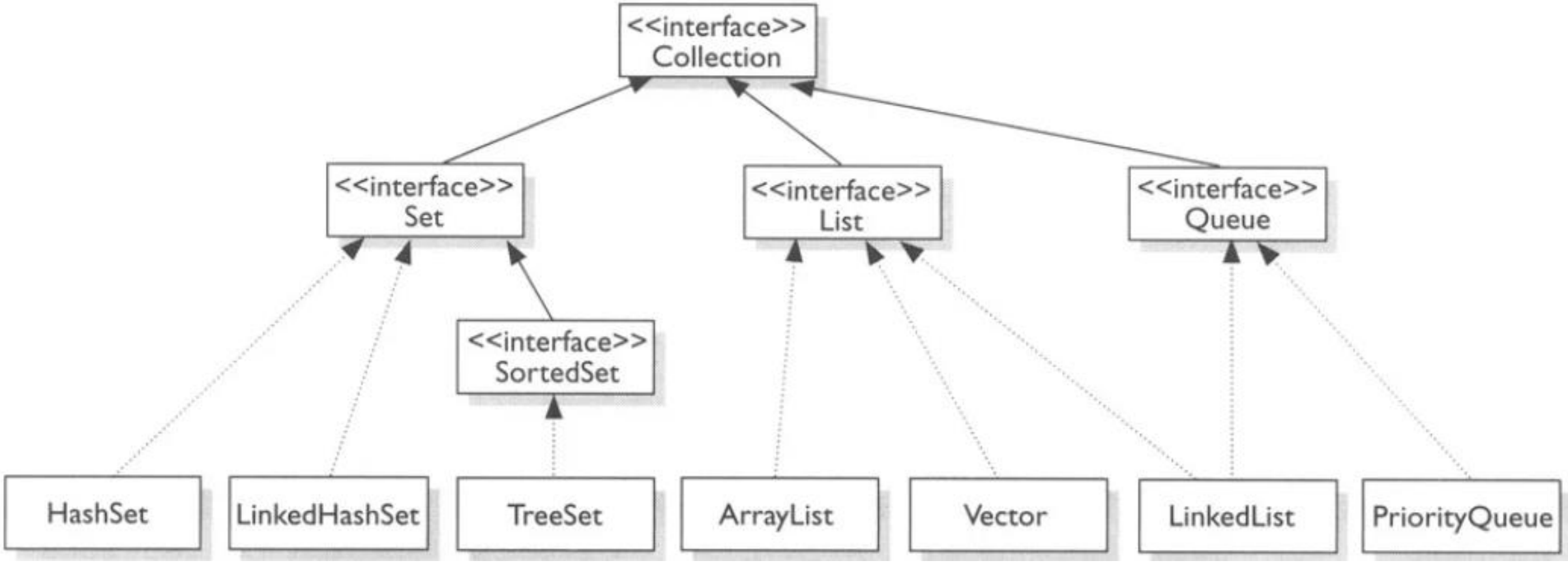
To comprehend how interfaces and classes are linked with each other and in what hierarchy they are situated, we will need to utilize a diagrams below.

Note Key.

Extends

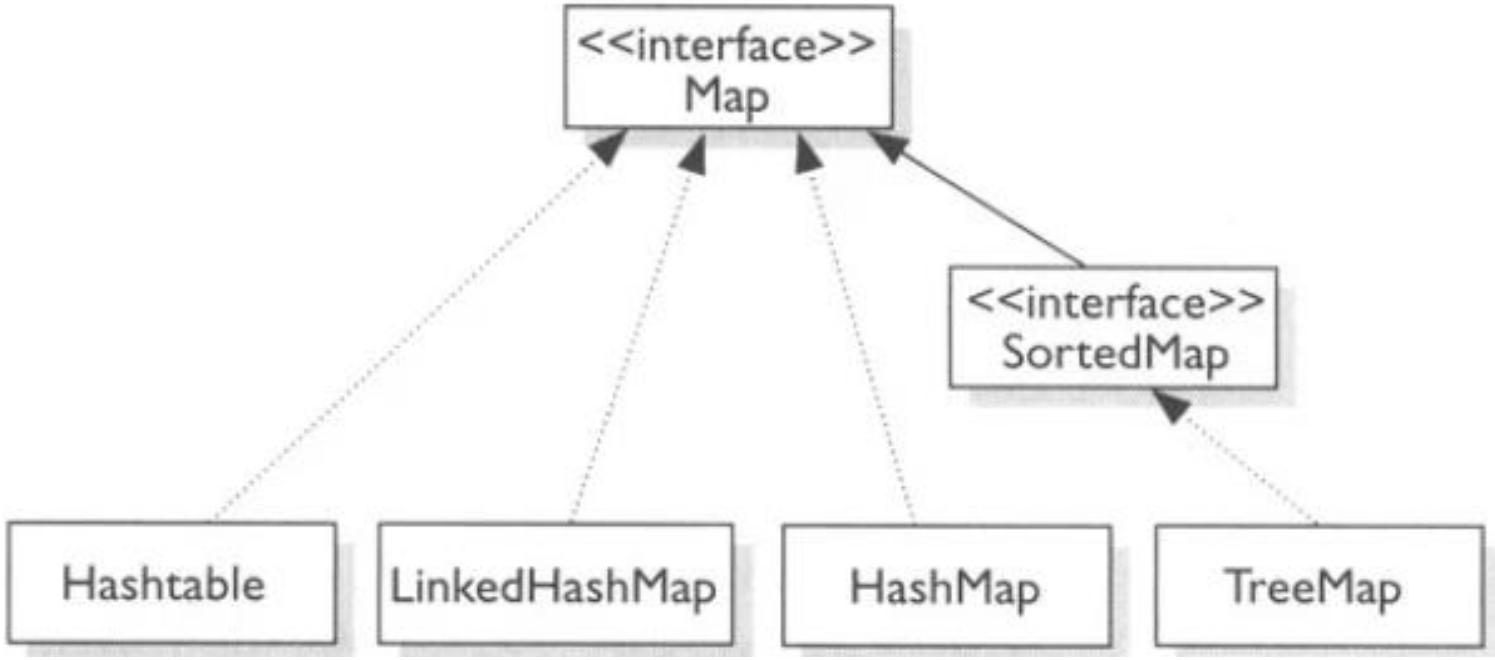
Implements

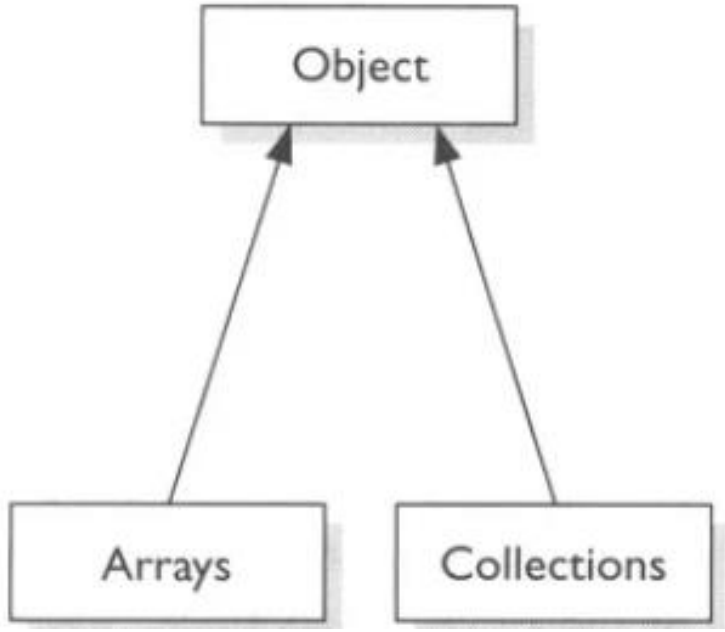
Java Collection Hierarchy +



(Studytonight.com)

Java Collection Hierarchy ++





(Studytonight.com)

Commonly Used Methods of Collection Framework

All these Interfaces operate based on several methods which are defined by collections classes which implement these interfaces

Method	**Description**
default boolean **removeIf**(Predicate<? super E> filter)	It deletes all the elements of the collection that satisfy the specified predicate.
public boolean **retainAll**(Collection<?> c)	It deletes all the elements of invoking collection except the specified collection.
public int **size**()	It returns the total number of elements in the collection.
public void **clear**()	It removes the total number of elements from the collection.
public boolean **contains**(Object element)	It searches for an element.
public boolean **containsAll**(Collection<?> c)	It is used to search the specified collection in the collection.

Commonly Used Methods of Collection Framework +

Method	**Description**
public boolean **add**(E e)	It inserts an element in this collection.
public boolean **addAll**(Collection<? extends E> c)	It inserts the specified collection elements in the invoking collection.
public boolean **remove**(Object element)	It deletes an element from the collection.
public boolean **removeAll**(Collection<?> c)	It deletes all the elements of the specified collection from the invoking collection.

Why Collections were made Generic?

Generics, added **type safety** to Collection framework. Earlier collections stored **Object class** references which meant any collection could store any type of object. Hence there were chances of storing incompatible types in a collection, which could result in run time mismatch.

Generics was introduced to **allow a programmer to explicitly state the type of object being stored.**

Collections and Autoboxing

We have studied that Autoboxing converts primitive types into Wrapper class Objects. As collections doesn't store primitive data types(stores only references), hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type.

Most Commonly thrown Exceptions in Collections Framework

There may be chance of getting exceptions while working with collections. We have listed some most common exceptions that may occur during program execution.

Exception Name	**Description**
UnSupportedOperationException	occurs if a Collection cannot be modified
ClassCastException	occurs when one object is incompatible with another
NullPointerException	occurs when you try to store null object in Collection
IllegalArgumentException	thrown if an invalid argument is used
IllegalStateException	thrown if you try to add an element to an already full Collection

Collection Interfaces

The Collection Interface

is at the top of collection hierarchy and must be implemented by any class that defines a collection. Its general declaration is,

```
interface Collection <E>
```

Collection Interface Methods

Following are some of the commonly used methods in this interface.

Methods	Description
boolean **add**(E obj)	Used to add objects to a collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
boolean **addAll**(Collection C)	Add all elements of collection C to the invoking collection. Returns true if the element were added. Otherwise, returns false.
boolean **remove**(Object obj)	To remove an object from collection. Returns true if the element was removed. Otherwise, returns false.
boolean **removeAll**(Collection C)	Removes all element of collection C from the invoking collection. Returns true if the collection's elements were removed. Otherwise, returns false.
boolean **contains**(Object obj)	To determine whether an object is present in collection or not. Returns true if obj is an element of the invoking collection. Otherwise, returns false.
boolean **isEmpty**()	Returns true if collection is empty, else returns false.

Collection Interface Methods+

Following are some of the commonly used methods in this interface.

int **size**()	Returns number of elements present in collection.
void **clear**()	Removes total number of elements from the collection.
Object[] **toArray**()	Returns an array which consists of the invoking collection elements.
boolean **retainAll**(Collection c)	Deletes all the elements of invoking collection except the specified collection.
Iterator **iterator**( )	Returns an iterator for the invoking collection.
boolean **equals**(Object obj)	Returns true if the invoking collection and obj are equal. Otherwise, returns false.
Object[] **toArray**(Object array[])	Returns an array containing only those collection elements whose type matches of the specified array.

The List Interface

extends the **Collection** Interface, and defines storage as sequence of elements. Following is its general declaration,

```
interface List <E>
```

1.Allows random access and insertion, based on position.

2.It also allows Duplicate elements.

List Interface Methods

Apart from methods of Collection Interface, List interface adds following methods of its own.

Methods	**Description**
Object **get**(int index)	Returns object stored at the specified index
Object **set**(int index, E obj)	Stores object at the specified index in the calling collection
int **indexOf**(Object obj)	Returns index of first occurrence of obj in the collection
int **lastIndexOf**(Object obj)	Returns index of last occurrence of obj in the collection
List **subList**(int start, int end)	Returns a list containing elements between start and end index in the collection

The Set Interface

This interface defines a Set. It extends **Collection** interface and doesn't allow insertion of duplicate elements.

Declaration

```
interface Set <E>
```

Features

1. It doesn't define any method of its own. It has two sub interfaces, **SortedSet** and **NavigableSet**.
2. **SortedSet** interface extends **Set** interface and arranges added elements in an ascending order.
3. **NavigabeSet** interface extends **SortedSet** interface, and allows retrieval of elements based on the closest match to a given value or values.

The Queue Interface

extends **collection** interface and defines behavior of queue, that is first-in, first-out.

It's general declaration is,

```
interface Queue <E>
```

Double ended queues can function as simple queues as well as like standard Stacks.

Queue Interface Methods

There are couple of new and interesting methods added by this interface. Some of them are mentioned in the table below .

Methods	**Description**
Object **poll**()	removes element at the head of the queue and returns **null** if queue is empty
Object **remove**()	removes element at the head of the queue and throws **NoSuchElementException** if queue is empty
Object **peek**()	returns the element at the head of the queue without removing it. Returns **null** if queue is empty
Object **element**()	same as peek(), but throws **NoSuchElementException** if queue is empty
boolean **offer**(E obj)	Adds object to queue.

The Dequeue Interface

It extends **Queue** interface and implements behavior of a double-ended queue. Its general declaration is,

```
interface Dequeue <E>
```

Collection Classes

The Collection classes in Java

Java collection framework consists of various classes that are used to store objects. These classes implements the Collection interface.

Some of them provide full implementations that can be used as it is. Others are abstract classes, which provides skeletal implementations that can be used as a starting point for creating concrete collections.

Java Collection Framework Classes +

The table below contains abstract and non-abstract classes that implements collection interface. The standard collection classes are:

Class	**Description**
AbstractCollection	Implements most of the Collection interface.
AbstractList	extends AbstractCollection and implements most of the List interface.
AbstractQueue	extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linkedList by extending AbstractSequentialList
ArrayList	Implements a dynamic array by extending AbstractList

Java Collection Framework Classes ++

This table contains abstract and non-abstract classes that implements collection interface.

The standard collection classes are:

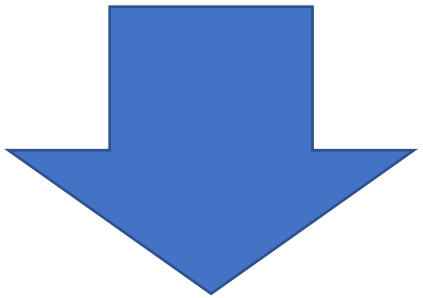
Class	Description
ArrayDeque	Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface(Added by Java SE 6).
AbstractSet	Extends AbstractCollection and implements most of the Set interface.
EnumSet	Extends AbstractSet for use with enum elements.
HashSet	Extends AbstractSet for use with a hash table.
LinkedHashSet	Extends HashSet to allow insertion-order iterations.
PriorityQueue	Extends AbstractQueue to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends AbstractSet.

Java Collection Framework Classes +++

Note:

1. To use any Collection class in your program, you need to import java.util package.
2. Whenever you print any Collection class, it gets printed inside the square brackets [] with its elements.

Lets look at how some of the collection classes operate



ArrayList class

provides implementation of an array based data structure that is used to store elements in linear order. This class **implements** List interface and **extends** AbstractList class. It creates a dynamic array that grows based on the elements strength.

Why use ArrayList Class?

Simple array has fixed size i.e it can store fixed number of elements but sometimes you may not know beforehand about the number of elements that you are going to store in your array. In such situations, We can use an ArrayList, which is an array whose size can increase or decrease dynamically.

ArrayList Class +

Features

1.ArrayList class extends **AbstractList** class and implements the **List** interface.

2.ArrayList supports dynamic array that can grow as needed.

3.It can contain Duplicate elements and it also maintains the insertion order.

4.Manipulation is slow because a lot of shifting of elements occurs if any element is removed from the array list.

5.ArrayLists are not synchronized.

6.ArrayList allows random access because it works on the index basis.

ArrayList Class +

ArrayList Constructors

ArrayList class has three constructors that can be utilized

```
ArrayList() // It creates an empty ArrayList
ArrayList( Collection C ) // creates an ArrayList
that is initialized with elements of the
Collection C
ArrayList( int capacity ) // creates an ArrayList
that has the specified initial capacity
```

ArrayList Class+++

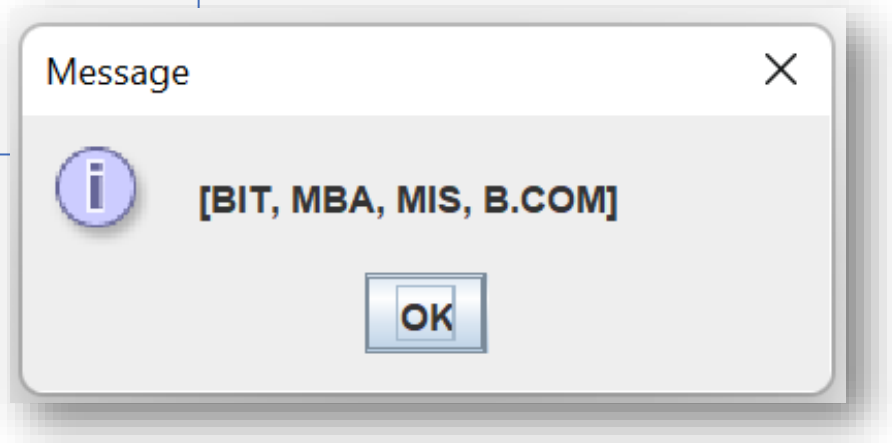
Example program

Lets create an ArrayList to store string elements. See, we used add method of list interface to add elements.

```
import java.util.*; import javax.swing.JOptionPane;
class AList {
      public static void main(String[] args) {
      ArrayList< String> al = new ArrayList< String>();
      al.add("BIT");
      al.add("MBA");
      al.add("MIS");
      al.add("B.COM");
      JOptionPane.showMessageDialog(null, al);
} }
```

ArrayList Class++++ Program Code+Output

```
1   package CollectionClass;
2   import java.util.*; import javax.swing.JOptionPane;
3   public class AList {
4       public static void main(String[] args) {
            ArrayList< String> al = new ArrayList< String>();
6           al.add("BIT");
7           al.add("MBA");
8           al.add("MIS");
9           al.add("B.COM");
10          JOptionPane.showMessageDialog(null, al);
11      }
12  }
```



LinkedList class

Java LinkedList class provides implementation of **linked-list data structure**.

Features

1. LinkedList class extends AbstractSequentialList and implements **List**, **Deque** and **Queue** interface.
2. It can be used as List, stack or Queue as it implements all the related interfaces.
3. It is dynamic in nature i.e it allocates memory when required. Therefore insertion and deletion operations can be easily implemented.
4. It can contain duplicate elements and it is not synchronized.
5. Reverse Traversing is difficult in linked list.
6. In LinkedList, manipulation is fast because no shifting needs to be occurred.

LinkedList class + Constructors

LinkedList class has two constructors.

```
LinkedList() // It creates an empty LinkedList
LinkedList( Collection c) // It creates a
LinkedList that is initialized with elements of
the Collection c
```

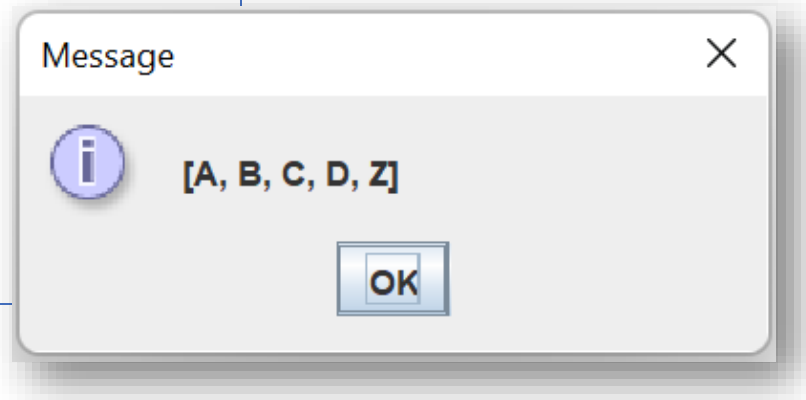
LinkedList class ++ Sample program

Lets take an example to create a linked-list and add elements using add and other methods as well. See the below example.

```
import java.util.* ; import javax.swing.JOptionPane;
class L_List {
      public static void main(String[] args) {
      LinkedList< String> L = new LinkedList< String>();
      L.add("B");
      L.add("C");
      L.add("D");
      L.addLast("Z");
      L.addFirst("A");
      JOptionPane.showMessageDialog(null, L);
} }
```

LinkedList class ++ Sample program code+Output

```
1   package CollectionClass;
2   import java.util.* ; import javax.swing.JOptionPane;
3   class L_List {
4       public static void main(String[] args) {
            LinkedList< String> L = new LinkedList< String>();
6           L.add("B");
7           L.add("C");
8           L.add("D");
9           L.addLast("Z");
10          L.addFirst("A");
11          JOptionPane.showMessageDialog(null, L);
12      }
13  }
```



Difference between ArrayList and LinkedList class

ArrayList and **LinkedList** are the Collection classes, and both of them implements the List interface. Following are their differences

No	ArrayList	LinkedList
1	allows random access to the elements in the list as it operates on an **index-based** data structure	does not allow random access as it does not have indexes to access elements directly, it has to traverse the list to retrieve or access an element from the list
2	extends AbstarctList class	extends AbstractSequentialList.
3	implements List interface, thus it can behave as a list only	implements List, Deque and Queue interface, thus it can behave as a Queue and List both.
4	access to elements is faster in ArrayList as random access is also possible	Access to LinkedList elements is slower as it follows sequential access only
5	manipulation of elements is slower in ArrayList	it is faster in LinkedList.

HashSet class

1. extends **AbstractSet** class and implements the **Set** interface.
2. It creates a collection that uses hash table for storage. A hash table stores information by using a mechanism called **hashing**. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored.
3. HashSet does not maintain any order of elements.
4. HashSet contains only unique elements.

This class has three constructors.

```
HashSet() //This creates an empty HashSet
HashSet( Collection C ) //This creates a HashSet that is initialized with the elements of the Collection C
HashSet( int capacity ) //This creates a HashSet that has the specified initial capacity
```

HashSet class +Example Program

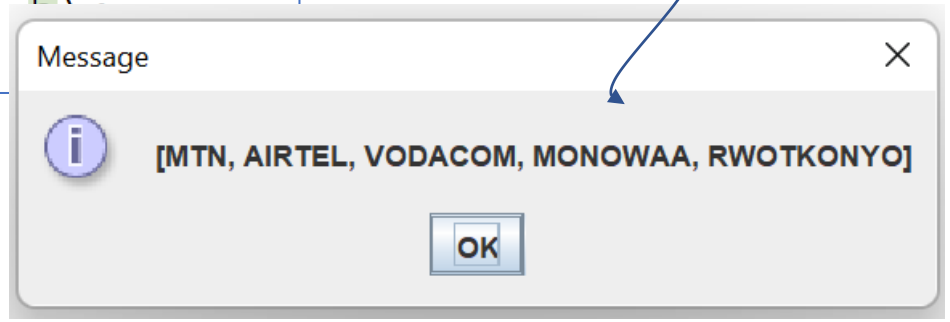
In this example, we are creating a HashSet that store string values. Since HashSet does not store duplicate elements, we tried to add a duplicate elements MONOWAA but the output contains only unique elements.

```
import java.util.* ; import javax.swing.JOptionPane;
class HSet{
      public static void main(String[] args) {
      HashSet< String> h = new HashSet< String>();
      h.add("MONOWAA");
      h.add("MTN");
      h.add("AIRTEL");
      h.add("VODACOM");
      h.add("MONOWAA"); h.add("RWOTKONYO");
      JOptionPane.showMessageDialog(null, h);
      }
}
```

HashSet class +Example Program code+Output

```
1   package CollectionClass;
2   import java.util.* ; import javax.swing.JOptionPane;
3   class HSet{
4       public static void main(String[] args) {
            HashSet< String> h = new HashSet< String>();
6           h.add("MONOWAA");
7           h.add("MTN");
8           h.add("AIRTEL");
9           h.add("VODACOM");
10          h.add("MONOWAA");
11          h.add("RWOTKONYO");
12          JOptionPane.showMessageDialog(null,
13      } }
```

NOTE! the duplicated item, MONOWAA is printed on once.



LinkedHashSet Class

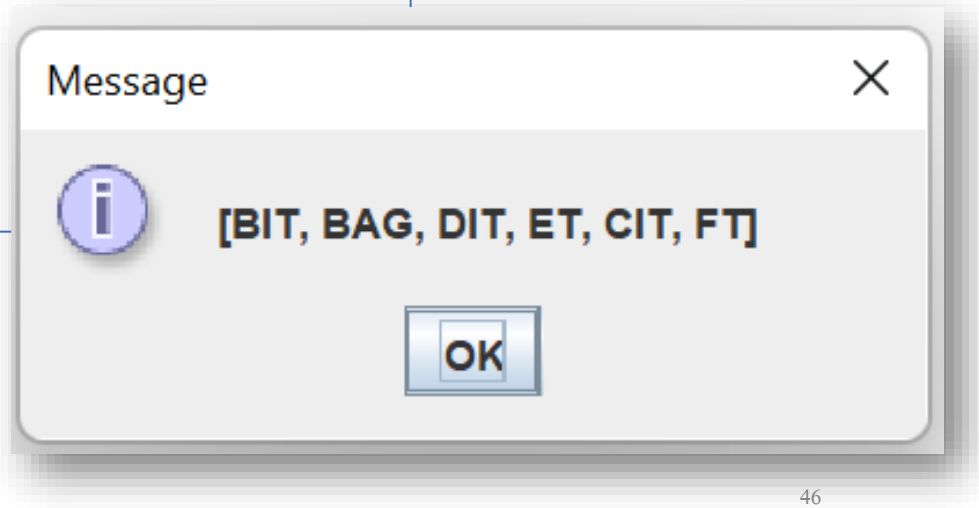
1. LinkedHashSet class extends **HashSet** class
2. LinkedHashSet maintains a linked list of entries in the set.
3. LinkedHashSet stores elements in the order in which elements are inserted i.e it maintains the insertion order.

LinkedHashSet Class+Example Program

```
import java.util.*; import javax.swing.JOptionPane;
class LHSet{
public static void main(String args[]) {
     LinkedHashSet<String>lhs = new LinkedHashSet<String>();
     lhs.add("BIT"); lhs.add("BAG");
     lhs.add("DIT"); lhs.add("ET");
     lhs.add("CIT"); lhs.add("FT");
     JOptionPane.showMessageDialog(null, lhs);
}
}
```

LinkedHashSet Class+Example Program code and Output

```
1   package CollectionClass;
2   import java.util.*; import javax.swing.JOptionPane;
3   class LHSet{
4   public static void main(String args[]) {
        LinkedHashSet<String>lhs = new LinkedHashSet<String>();
6       lhs.add("BIT"); lhs.add("BAG");
7       lhs.add("DIT"); lhs.add("ET");
8       lhs.add("CIT"); lhs.add("FT");
9       JOptionPane.showMessageDialog(null, lhs);
10  }
11  }
```



TreeSet Class

1.extends **AbstractSet** class and implements the **NavigableSet** interface.

2.stores the elements in ascending order.

3.uses a Tree structure to store elements.

4.contains unique elements only like HashSet.

5.It's access and retrieval times are quite fast.

This class has 4 Constructors.

TreeSet Class+Constructor

This class has Four Constructors.

```
TreeSet() //creates an empty tree set that will be sorted in an
ascending order according to the natural order of the tree set

TreeSet( Collection C ) //creates a new tree set that contains the
elements of the Collection Class

TreeSet( Comparator comp ) //creates an empty tree set that will be
sorted according to given comparator

Comparator TreeSet( SortedSet ss ) //creates a TreeSet that contains
the elements of given SortedSet
```

TreeSet Class+Example Program

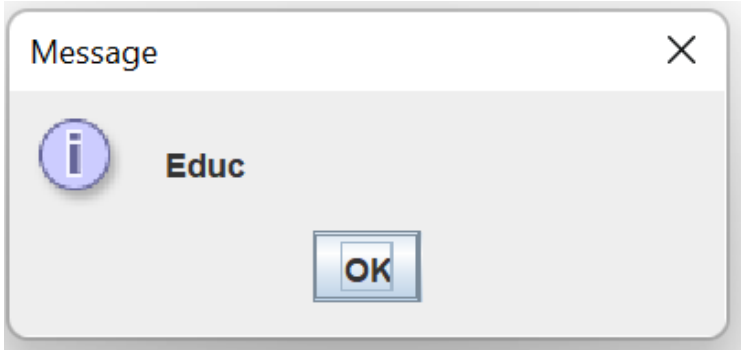
In this example, we are creating a treeset that contains duplicate elements. But you can **notice** that it prints unique elements that means it does not allow duplicate elements.

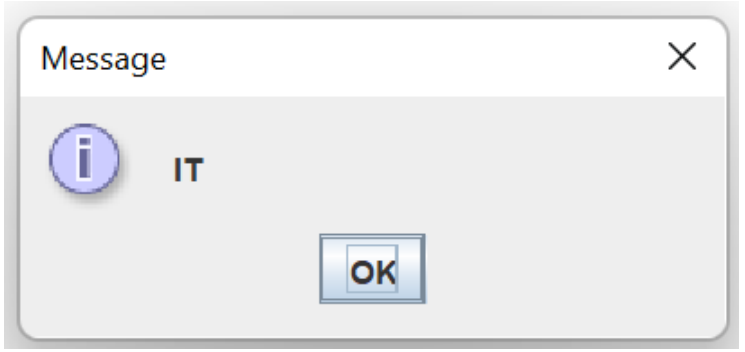
```
import java.util.*; import javax.swing.JOptionPane;
class Tset{
 public static void main(String args[]){
      TreeSet<String> ts=new TreeSet<String>();
      ts.add("IT"); ts.add("Educ");
      ts.add("Educ"); ts.add("SOS");
      Iterator loop = ts.iterator();
      while(loop.hasNext()){
            JOptionPane.showMessageDialog(null, loop.next());
      }
} }
```

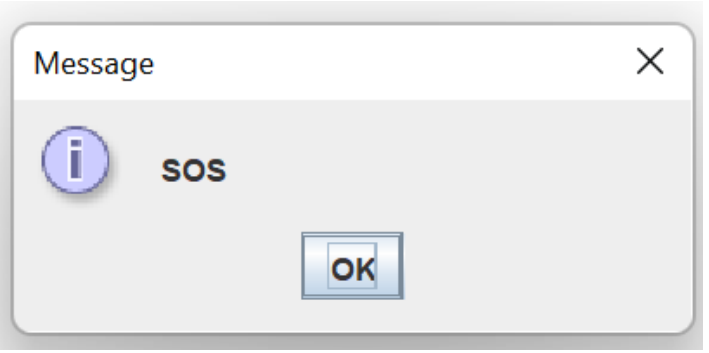
TreeSet Class Example Program code+Output

```
1   package CollectionClass;
2   import java.util.*; import javax.swing.JOptionPane;
3   class Tset{
4    public static void main(String args[]){
5        TreeSet<String> ts=new TreeSet<String>();
6        ts.add("IT"); ts.add("Educ");
7        ts.add("Educ"); ts.add("SOS");
8        Iterator loop = ts.iterator();
9        while(loop.hasNext()){
10           JOptionPane.showMessageDialog(null, loop.next());
11           }
12   } }
```

Note that Educ could not be printed twice although each item is printed on JOptionPane one at a time on the click of OK Button.







Iterator and ListIterator

Iterator is an interface that is used to iterate the collection elements. It is part of java collection framework. It provides some methods that are used to check and access elements of a collection.

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide iterator() method to return an iterator.

Accessing a Java Collection using Iterators

To access elements of a collection, we can either use index if collection is list based or we need to traverse the element. There are three possible ways to traverse through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

Iterator Interface Methods

Method	**Description**
boolean hasNext()	Returns **true** if there are more elements in the collection. Otherwise, returns false.
E next()	Returns the **next element present** in the collection. Throws NoSuchElementException if there is not a next element.
void remove()	Removes the current element. Throws IllegalStateException if an attempt is made to call remove() method that is not preceded by a call to next() method.

Iterator Example Program

In this example, we are using iterator() method of collection interface that returns an instance of Iterator interface. After that we are using hasNext() method that returns true if collection contains an elements within the loop, obtain each element by calling next() method.

```
import java.util.*;
class ITPro {
public static void main(String[] args) {
      ArrayList< String> al = new ArrayList< String>();
      al.add("BIT");
      al.add("CIT");
      al.add("ET");
      al.add("DIT");
      Iterator it = al.iterator(); //Declaring Iterator
      while(it.hasNext()) {
      System.out.print(it.next()+" ");
      }
} }
```

Iterator Example Program code+output

```
1  package CollectionClass; import java.util.*;
2  class ITPro {
3      public static void main(String[] args) {
           ArrayList< String> al = new ArrayList< String>();
5          al.add("BIT");
6          al.add("CIT");
7          al.add("ET");
8          al.add("DIT");
9          Iterator it = al.iterator(); //Declaring Iterator
10         while(it.hasNext()) {
11         System.out.print(it.next()+" ");
12         }
13     } }
```

```
run:
BIT CIT ET DIT
```

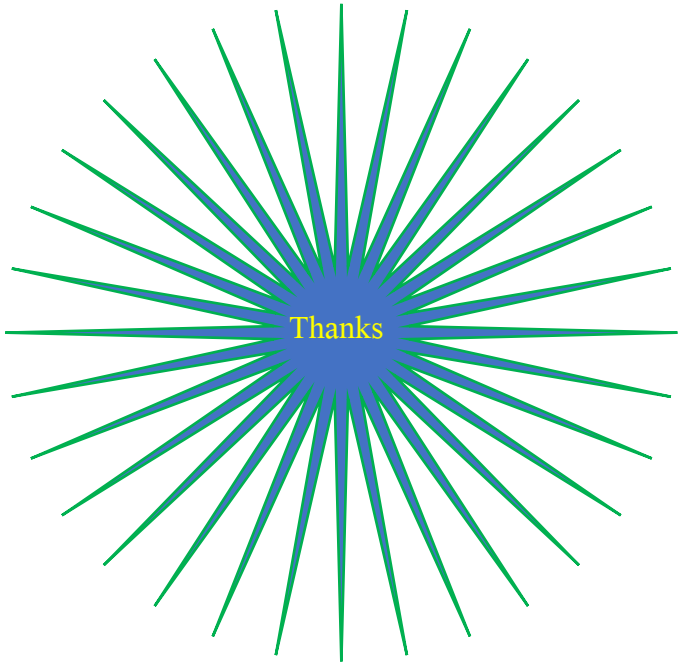
NOTE: More about Iterator and ListItierator will be share

in the lecture Lecturer

Summary

1. Introduction to Collection framework,
2. Collection Interfaces,
3. Collection Classes,
4. Iterator and ListIterator

Thank you for Listening



References

Java Collection Framework. Studytonight.com. (n.d.). Retrieved October 28, 2022, from https://www.studytonight.com/java/collection-framework.php