* Hierarchy of Collection Framework
* ArrayList class
* LinkedList class
* List Interface
* HashSet class

**The limitations of arrays:**

1.arrays are fixed in length.

Once an array is created we can neither increase or decrease the length of the array.

2.arrays hold homogeneous data elements.

3.There is no underlying data structure for arrays, i.e., there are no readymade method support for arrays.

**We can overcome the above limitations if we go for collections**.

1.Collections are growable in nature, size can be increase/decreased dynamically.

2.Collections can hold heterogeneous objects.

3.Every collection is based on some data structure.

Collection: if we want to represent group of objects as a single entity then we can go use collection framework.

In the collection frame work the root interface is Collection interface.

Collection(Interface)

1.5

List(Interface) Set(interface) Queue(interface)

ArrayList(c) LinkedList(c) Vector(c) HashSet(C) SortedSet(I) PriorityQueue BlockingQ

1.2

1.0

1.4

1.2

Stack(c) LinkedHashSet(c) NavigableSet(I)

1.4

1.0

TreeSet(C)

1.5

If we want to represent group of objects where duplicates are allowed and insertion order is preserved then we can use List object.

Ex:

**package** pack1;

**import** java.util.ArrayList;

**publicclass** ArrayListEx {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

ArrayListal=**new**ArrayList();

al.add("hyderabad");

al.add("pune");

al.add(**new** Integer(10));

al.add("pune");

System.***out***.println(al);//[hyderabad, pune, 10, pune]

}

}

Ex2:

**package** pack1;

**import** java.util.ArrayList;

**publicclass** ArrayListEx {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

ArrayListal=**new**ArrayList();

al.add("hyderabad");

al.add("pune");

al.add(**new** Integer(10));

al.add("pune");

System.***out***.println(al);//[hyderabad, pune, 10, pune]

//without generics

Object o1=al.get(3);

**if** (o1**instanceof** String)

{

String s=(String)o1;

System.***out***.println(s);

}

**elseif** (o1**instanceof** Integer)

{

Integer I=(Integer)o1;

System.***out***.println(I);

}

}

}

Ex3:using generics

**package** pack1;

**import** java.util.ArrayList;

**publicclass** ArrayListEx {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

//with generics

ArrayList<String>al=**new** ArrayList<String>();

al.add("hyderabad");

al.add("pune");

//al.add(new Integer(10)); It gives CE

al.add("pune");

System.***out***.println(al);//[hyderabad, pune, pune]

String s1=al.get(2);

System.***out***.println(s1);

System.***out***.println(al.get(2));

}

}

output:

[hyderabad, pune, pune]

pune

pune

if we want to represent group of objects where duplicates are not allowed and insertion order is not preserved then use HashSet.

If we want to represent group of objects where duplicates are not allowed but insertion order is preserved then use LinkedHashSet.

**package** pack1;

**import** java.util.HashSet;

**publicclass** HashSetEx {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

HashSet<Integer>hs=**new** HashSet<Integer>();

hs.add(10);

hs.add(20);

hs.add(30);

hs.add(40);

System.***out***.println(hs);//[20, 40, 10, 30]

hs.add(10);//no CE, no RE but it will not add duplicate values

}

}

**Cursors/Iterators :**

**Cursors are used to iterate through the elements of collection objects.**

**There are 3 types of cursors**

1. **Enumeration**
2. **Iterator**
3. **ListIterator**

**Enumeration is used to iterate through the elements of legacy classes like Vector, Properties classes.**

**Ex1:**

**package** pack1;

**import** java.util.Enumeration;

**import** java.util.Vector;

**publicclass** TestEnumeration {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

Vector<Integer>v=**new** Vector<Integer>();

**for** (**int**i=1;i<=10;i++)

{

v.addElement(i);

}

System.***out***.println(v);

Enumeration<Integer>e=v.elements();

**while**( e.hasMoreElements())

{

Integer I=e.nextElement();

**if** (I%2==0)

System.***out***.println(I);

}

}

}

the limitation of Enumeration is that it can be used only for legacy classes(old classes) like Vector, Properties.

It cannot be used for other classes like ArrayList.

Using Enumeration we can only read, we cannot remove.

To overcome these limitations we need to use Iterator.

Ex:

**package** pack1;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**publicclass** TestEnumeration {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

ArrayList<Integer>al=**new** ArrayList<Integer>();

**for**(**int**i=1;i<=10;i++)

{

al.add(i);

}

Iterator<Integer>Itr=al.iterator();

**while**(Itr.hasNext())

{

Integer I=Itr.next();

**if** (I%2==0)

System.***out***.println(I);

**else**

Itr.remove();

}

System.***out***.println(al);

}

}

output:

2

4

6

8

10

[2, 4, 6, 8, 10]

Limitations of Iterator:

i>though it can be used for every collection object, using Iterator we can move only in forward direction we cannot move in reverse direction.

ii> using Iterator we can only remove elements, we cannot add or replace.

We can resolve these problems using ListIterator.

Ex:

**package** pack1;

**import** java.util.LinkedList;

**import** java.util.ListIterator;

**publicclass** TestEnumeration {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

LinkedList<String>ll=**new** LinkedList<>();

ll.add("sriram");

ll.add("ravikanth");

ll.add("kosmik");

ll.add("vision");

System.***out***.println(ll);

ListIterator<String>litr=ll.listIterator();

**while**(litr.hasNext())

{

String s=litr.next();

**if** (s.equals("kosmik"))

litr.set("bigclasses");

}

System.***out***.println(ll);

}

}

Output:

[sriram, ravikanth, kosmik, vision]

[sriram, ravikanth, bigclasses, vision]

Note: ListIterator can be used only for List implemented classes like HashSet, LinkedHashSet and TreeSet. Not for any other classes.

Set implemented classes:

Ex on HashSet

**package** pack1;

**import** java.util.HashSet;

**publicclass** HashSetEx1 {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

HashSet<String>hs=**new** HashSet<>();

hs.add("hyd");

hs.add("secbad");

hs.add("kphb");

System.***out***.println(hs);

hs.add("hyd");//No CE, No RE but the element will not be added

}

}

Ex2:

**package** pack1;

**import** java.util.HashSet;

**import** java.util.Iterator;

**publicclass** HashSetEx1 {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

HashSet<String>hs=**new** HashSet<>();

hs.add("hyd");

hs.add("secbad");

hs.add("kphb");

System.***out***.println(hs);//[hyd, kphb, secbad]

hs.add("hyd");//No CE, No RE but the element will not be added

Iterator<String>itr=hs.iterator();

**while**(itr.hasNext())

{

String s=itr.next();

**if**(s.equals("kphb"))

itr.remove();

}

System.***out***.println(hs);//[hyd, secbad]

}

}

TreeSet:

If we are adding objects and if we are depending upon default natural sorting order then the objects being added should be homogeneous and comparable.

An object is said to be comparable if the corresponding class implements Comparable interface.

String class and Integer class implements Comparable interface.

Ex:

If we depend upon default sorting order, then add comparable objects directly.

**package** pack1;

**import** java.util.TreeSet;

**publicclass** TreeSetEx1 {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

TreeSet<Integer>ts=**new** TreeSet<>();

ts.add(10);

ts.add(90);

ts.add(30);

ts.add(70);

ts.add(20);

System.***out***.println(ts);

}

}

Ex2:if we want descending order of numbers then we need to implement Comparator interface.

**package** pack1;

**import** java.util.Comparator;

**import** java.util.TreeSet;

**publicclass** TreeSetEx1 {

**publicstaticvoid** main(String[] args) {

// **TODO** Auto-generated method stub

TreeSet<Integer>ts=**new** TreeSet<>(**new** MyComparator());

ts.add(10);

ts.add(90);

ts.add(30);

ts.add(70);

ts.add(20);

System.***out***.println(ts);

}

}

**class** MyComparator **implements**Comparator

{

@Override

**publicint**compare(Object o1, Object o2) {

// **TODO** Auto-generated method stub

Integer I1=(Integer)o1;

Integer I2=(Integer)o2;

**return** -I1.compareTo(I2);

}

}

Output:

[90, 70, 30, 20, 10]