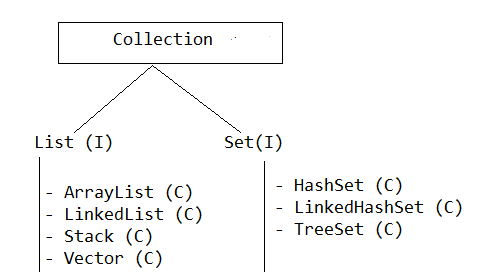
**Collection Framework**

* Collection framework provides set of classes and interfaces that are used to represent group of objects as a single entity.

**Collection Framework Hierarchy**



**List Interface**

* List interface has following proprieties
  + Allows null insertion.
  + Allows duplicate objects.
  + Preserved insertion order

**ArrayList**

* ArrayList stores Heterogeneous objects.
* ArrayList allowed null insertion.
* ArrayList preserved Insertion order.
* Duplicate objects are allowed.
* The under laying data structure is growable array.
* Every method present in the ArrayList is not synchronized

**Example**

**import** java.util.ArrayList;

**public** **class** ArrayListDemo {

**public** **static** **void** main(String[] args) {

ArrayList al = **new** ArrayList();

al.add(10);

al.add(20);

al.add("bbsr");

al.add(**null**);

al.add(20);

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

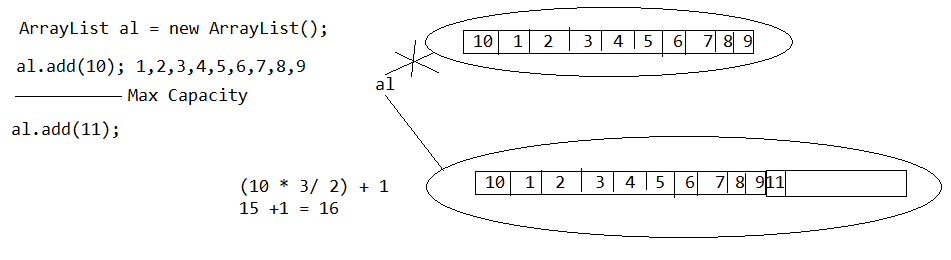
}

}

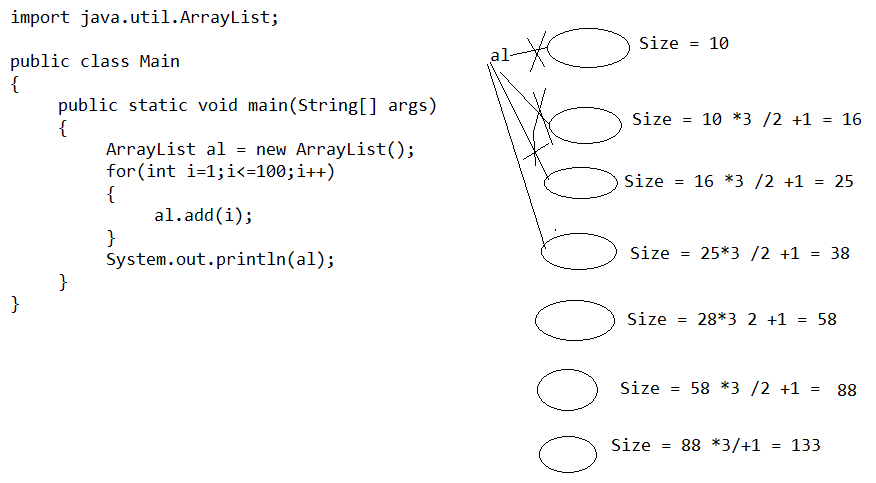
**Note**

* The default capacity of the ArrayList is **10** once it reaches its maximum capacity then size is automatically increased by:

**New capacity = ((old capacity\*3)/2 )+1**



**Example**



import java.util.ArrayList;

public class Main

{

public static void main(String[] args)

{

ArrayList al = new ArrayList(100);

for(int i=1;i<=100;i++)

{

al.add(i);

}

System.out.println(al);

}

}

**Example**

**import** java.util.ArrayList;

**public** **class** ArrayListDemo {

**public** **static** **void** main(String[] args) {

ArrayList al = **new** ArrayList();

al.add("A");

al.add("B");

System.***out***.println(al); //[A, B]

al.add(1,"C");

System.***out***.println(al);// [A, C, B]

al.remove("B");

System.***out***.println(al); // [A, C]

al.remove(0);

System.***out***.println(al);// [C]

}

}

**Generic version of ArrayList**

**import** java.util.ArrayList;

**public** **class** ArrayListDemo {

**public** **static** **void** main(String[] args) {

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

al.add("Ram");

al.add("Raj");

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

}

}

**LinkedList**

* Heterogeneous objects are allowed.
* Null insertion is possible.
* Insertion order is preserved
* Duplicate objects are allowed.
* The under laying data structure is double linked list.
* Every method present in the ArrayList is not synchronized

**Example**

**import** java.util.LinkedList;

**public** **class** LinkedListDemo {

**public** **static** **void** main(String[] args) {

LinkedList list = **new** LinkedList();

list.add(10);

list.add(30);

list.add("bbsr");

list.add(**null**);

list.add(30);

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

}

}

**Note**

* In ArrayList, if we are adding/removing object at the middle of ArrayList then number of shift operations are requires. Hence ArrayList is not suitable for insertion and deletion operation.
* LinkedList is recommended to perform insertion and deletion operation.
* ArrayList is suitable for fetch operation but LinkedList is not suitable.

**Vector**

* Heterogeneous objects are allowed
* Null insertion is possible
* Duplicate objects are allowed
* Insertion order is preserved
* The under laying data structure is growable array
* Every method present in the Vector is synchronized

**Example**

**import** java.util.Vector;

**public** **class** VectorDemo {

**public** **static** **void** main(String[] args) {

Vector v = **new** Vector();

v.add(10);

v.add(20);

v.add("BBSR");

v.add(**null**);

v.add("BBSR");

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

}

}

**Note**

* The default capacity of the vector is 10 once it reaches its maximum capacity then size is automatically increased by:

New capacity = current capacity\*2

**Example**

**import** java.util.Vector;

**public** **class** VectorDemo {

**public** **static** **void** main(String[] args) {

Vector v = **new** Vector();

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

{

v.add(i);

}

System.***out***.println("Capacity = "+v.capacity());

System.***out***.println("Size = "+v.size());

v.add(11);

System.***out***.println("Capacity = "+v.capacity());

System.***out***.println("Size = "+v.size());

}

}

**Stack**

* It is a child class of vector.
* It is designed for LIFO (last in fist order )

**Example**

**import** java.util.Stack;

**public** **class** StackDemo {

**public** **static** **void** main(String[] args) {

Stack s = **new** Stack();

s.push(10);

s.push(20);

s.push(30);

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

s.pop();

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

}

}

**HashSet**

* HashSet stores Heterogeneous objects.
* HashSet allowed null insertion.
* HashSet not preserved Insertion order.
* Duplicate objects are not allowed.
* The under laying data structure is hash table.
* Every method present in the HashSet is not synchronized.

**Example**

**import** java.util.HashSet;

**public** **class** HashSetDemo {

**public** **static** **void** main(String[] args) {

HashSet hs = **new** HashSet();

hs.add(10);

hs.add(20);

hs.add("bbsr");

hs.add(**null**);

hs.add("bbsr");

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

}

}

**LinkedHashSet**

* HashSet stores Heterogeneous objects.
* HashSet allowed null insertion.
* LinkedHashSet preserved Insertion order.
* Duplicate objects are not allowed.
* The under laying data structure is hash table and linked list.
* Every method present in the HashSet is not synchronized.

**Example**

**import** java.util.LinkedHashSet;

**public** **class** LinkedHashSetDemo {

**public** **static** **void** main(String[] args) {

LinkedHashSet hs = **new** LinkedHashSet();

hs.add(10);

hs.add(20);

hs.add("bbsr");

hs.add(**null**);

hs.add("bbsr");

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

}

}

**TreeSet**

* TreeSet does not allowed heterogeneous object.
* TreeSet class doesn't allow null element.
* TreeSet class does not allowed duplicate value.
* Every methods present in TreeSet are not synchronized.
* TreeSet class maintains ascending order.

**Example**

**import** java.util.TreeSet;

**public** **class** TreeSetDemo {

**public** **static** **void** main(String[] args) {

TreeSet ts = **new** TreeSet();

ts.add(10);

ts.add(30);

ts.add(20);

ts.add(30);

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

}

}

**HashMap**

* It used to hold key value pairs.
* Duplicate keys are not allowed but values can be duplicated.
* Insertion order is not preserved.
* Null is allowed for key (only once)and allows for values any number of times.
* Underlying data Structure is Hashtable.
* Every method is non-synchronized.

**Example**

import java.util.\*;

class HashMapDemo

{

public static void main(String[] args)

{

HashMap h=new HashMap();

h.put("Android",7000);

h.put("Core Java",5000);

h.put("PHP",7000);

System.out.println(h);

}

}

**LinkedHashMap**

* Underlying data Structure is Hash Table & LinkedList.
* Duplicate keys are not allowed but values can be duplicated.
* Insertion order is preserved.

**Example**

import java.util.\*;

class LinkedHashMapDemo

{

public static void main(String[] args)

{

LinkedHashMap h=new LinkedHashMap();

h.put("Android",7000);

h.put("Core Java",5000);

h.put("PHP",7000);

System.out.println(h);

}

}

**Cursor**

* Cursor is used to retrieve objects one by one from the collection.
* There are three types of cursors are available in java
  + Enumeration
  + Iterator
  + ListIterator

**Enumeration**

* We can use Enumeration to get objects one by one from the old collection objects(e.g. vector, stack).

**Example**

import java.util.\*;

public class Main

{

public static void main(String[] args)

{

Vector v = new Vector();

for(int i=0;i<=10;i++)

{

v.add(i);

}

System.out.println(v);

Enumeration e = v.elements();

while(e.hasMoreElements())

{

Integer i = (Integer)e.nextElement();

System.out.println(i);

}

}

}

**Iterator**

* We can apply Iterator concept for any Collection object hence it is universal cursor.
* By using Iterator we can perform both read and remove operations.

**Example**

import java.util.\*;

public class Main

{

public static void main(String[] args)

{

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

for(int i=0;i<10;i++)

{

al.add(i);

}

System.out.println(al);

Iterator itr = al.iterator();

while(itr.hasNext())

{

Integer i = (Integer)itr.next();

System.out.println(i);

}

}

}

**Example**

import java.util.\*;

public class Main

{

public static void main(String[] args)

{

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

for(int i=0;i<10;i++)

{

al.add(i);

}

System.out.println(al);

Iterator itr = al.iterator();

while(itr.hasNext())

{

Integer i = (Integer)itr.next();

if(i%2==0)

itr.remove();

}

System.out.println(al);

}

}

**ListIterator**

* By using ListIterator we can move either to the forward direction or to the backward direction.
* By using ListIterator we can perform addition, replacement ,read and remove operations.

**Method of List Iterator**

* Forward direction
  + public boolean hasNext()
  + public void next()
  + public int nextIndex()
* Backward direction
  + public boolean hasPrevious()
  + public void previous()
  + public int previousIndex()
* Other capability method
  + public void remove()
  + public void set(Object new)
  + public void add(Object new)

**Example**

**import** java.util.ArrayList;

**import** java.util.ListIterator;

**public** **class** ListIteratorDemo {

**public** **static** **void** main(String[] args) {

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

al.add("bbsr");

al.add("pune");

al.add("cdac");

ListIterator li = al.listIterator();

System.***out***.println("Forward Direction : ");

**while**(li.hasNext()) {

String s = (String)li.next();

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

}

System.***out***.println("Backward Direction : ");

**while**(li.hasPrevious()) {

String s = (String)li.previous();

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

}

}

}

