Limitations of Arrays
++++++++++++++++++++++
1. fixed size, can't increase nor decrease.
2. it can't hold hetergenous elements.
3. it doesn't follow any datastructure, so ready made methods are not available to perform specific task.

Solution : Go for Collections

Benefits of Collections
+++++++++++++++++++++++
1. It is dynamic in nature(can grow in size)
2. it can hold both homogenous and heteregenous objects.
3. Every collection object is implemented using standard datastructure, so ready made support of methods are avaiable
so progrmaming is easy.

Which one is good either collections or Arrays?
Ans. Performance :: Arrays are recomended.
:: Collections are not recomended.

memory :: Arrays are not recomended.
:: Collections are recomended.

Difference b/w Collection vs Arrays
=> Arrays -> it can hold both primtive type data and object type data.
Collection -> it can hold only Objects type of data.

What is Collection?
In Order to represent a group of individual object as a single entity then we need to go for Collection.

CollectionFramework
Group of classes and interface, which can be used to represent group of individual object as a single entity, then we
need to go for "CollectionFramework".

Java		C++
Collection	=>	container
CollectionFramework	=>	STL(standard template library)

To know more information about the framework,then we need to know the specification(interface)

9 key interfaces of Collection framework
a. Collection(I)
b. List(I)
c. Set(I)
d. SortedSet(I)
e. NavigableSet(I)
f. Queue(I)
g. Map(I)
h. SortedMap(I)
i. NavigableMap(I)

Collection
1. In order to represent a group of individual object, then we need to go for "Collection".
2. It is a root interface of collection framework

3. All the commonly used method required for all the collection is a part of Collection(I).

Note: There is no concrete class which would implement the interface Collection(I) directly.

Difference b/w Collection(I) and Collections(C)?
Collection => It is an interface which should be used when we want to represent a group of individual object then we need
                to go for collection.

Collections => It is a utility class which defines in java.util which defines utility methods for Collection Objects.

List(I)
1. Insertion order must be preserved.
2. Duplicates are allowed.
3. It is the child interface of Collection.

Vector and Stack are a part of jdk1.0 version so they are called as "legacy classes".

Set(I)
It is used to represent a group of individual objects as a single entity such that
1. Duplicates are not allowed
2. Insertion is not preserved
3. It is the child interface of "Collection",
   then we need to go for "Set".

SortedSet(I)
It is used to represent a group of individual objects as a single entity such that
1. Duplicates are not allowed
2. elements should be added based on some sorting order
3. It is the child interface of "Set".
   then we need to go for "SortedSet".

NavigableSet(I)
1. It is a child class of SortedSet.
2. Various methods are a part of NavigableSet for navigation purpose.
3. Implementation class for NavigableSet is TreeSet.

What is the difference b/w List and Set?
List => Duplication are allowed, insertion order is preserved.
Set => Duplication are not allowed, insertion order not preserved.

Queue (I):
=> It is the Child Interface of Collection.
=> If we want to Represent a Group of Individual Objects Prior to Processing then we should go for Queue.

Eg: Before sending a Mail we have to Store All MailID’s in Some Data Structure and in which Order we added MailID’s in the Same Order
    Only Mails should be delivered (FIFO). For this Requirement Queue is Best Suitable.

Map(I)
  a. To represent a group of individual objects as keyvalue pair then we need to opt for Map(I).

8) SortedMap (I):
=> It is the Child Interface of Map.
=> If we want to Represent a Group of Objects as Key- Value Pairs according to Some Sorting Order of Keys then we should go for SortedMap.
=> Sorting should be Based on Key but Not Based on Value

9) NavigableMap (I):
=> It is the Child Interface of SortedMap.
=> It Defines Several Methods for Navigation Purposes.

Legacy Characters
 1. Enumeration
 2. Dictionary
 3. Vector
 4. Stack
 5. Hashtable
 6. Properties

Utility classes
 1. Collections
 2. Arrays

Cursors[Used for traversing/iterating the objects in collection]
 1. Enumeration(I)
 2. Iterator(I)
 3. ListIterator(I)

Sorting
 1.Comparable(I)
 2.Comparator(I)

Collection(I)
=============
  1. Inside this interface, the commonly used method required for all the collection classes is present

a. boolean add(object o)
       Only one object

b. boolean addAll(Collection c)
      To add group of Object

c. boolean remove(Object o)
      To remove particular object

d. boolean removeAll(Collection c)
      To remove particular group of collection

e. void clear()
      To remove all the object

f. int size()

To check the size of the array

g. boolean retainAll(Collection c)
except this group of objects remaining all objects should be removed.

h. boolean contains(Object o)
To check whether a particular object exists or not

i. boolean containsAll(Collection c)
To check whether a particular Collection exists or not

j. boolean isEmpty()
To check whether the Collection is empty or not

k. Object[] toArray()
Convert the object into Array.

l. Iterator iterator()
cursor need to iterate the collection object

Note :There is no concrete class which implements Collection interface directly.

List(I)
1. It is the child interface of Collection
2. To represent the group of collection objects where
   a. duplicates are allowed(meaning it is stored in index)
   b. insertion order is preserved.(meaning it is stored via index)
3. In list index plays a very important role

Method assoicated with List(I)
================================
a. void add(int index,Object obj)
b. void addAll(int index,Collection c)
c. Object remove(int index)
d. Object get(int index)
e. Object set(int index,Object o)
f. int indexOf(Object obj)
g. int lastIndexOf(Object obj)
i. ListIterator listIterator()

ArrayList(C)
===========
1. DataStructure: GrowableArray /Sizeable Array
2. Duplicates are allowed through index
3. insertion order is preserved through index
4. Heterogenous objects are allowed.
5. null insertion is also possible.

Constructors
===========
a. ArrayList al=new ArrayList()
Creates an empty ArrayList with the capacity to 10.
a. if the capacity is filled with 10, then what is the new capacity?
newcapacity= (currentcapacity * 3/2)+1
so new capacity is =16,25,38,.....
b. if we create an ArrayList in the above mentioned order then it would result in performance issue.
c. To resolve this problem create an ArrayList using 2nd way apporach.

b. ArrayList al=new ArrayList(int initalCapacity)

c. ArrayList l=new ArrayList(Collection c)
     It is used to create an equivalent ArrayList Object based on the Collection Object

eg#1.
```
import java.util.*;

public class Test
{
      public static void main(String[] args)
      {
            //List : index plays a role[internally Array]
            ArrayList l = new ArrayList();
            System.out.println("The size of list is :: "+l.size());

            l.add("A");
            l.add(10);
            l.add("A");
            l.add(null);

            System.out.println("The size of list is :: "+l.size());
            l.remove(2);
            System.out.println("The size of list is :: "+l.size());
            System.out.println(l);

            l.add(2,"sachin");
            System.out.println(l);
      }
}
```
Output
```
The size of list is :: 0
The size of list is :: 4
The size of list is :: 3
[A, 10, null]
[A, 10, sachin, null]
```

Note: Whenever we print any reference it internally calls toString() method.
      toString() of all Collection classes is implemented in such a way that it prints the Object
      in the following order.
         o/p => [,,,]
      toString() of all Map Object is implemented in such a way that it prints the Object in the
      following order.
        o/p => {k1=v1,k2=v2,k3=v3,....}

Usally we use Collection to store mulitple objects into single entity.
  Collection => contanier
To transport the collection over the network,compulsorily the Object should be "Serializable".
 1. Every Collection class by default implements Serializable.
 2. Every Collection class by default implements Cloneable

ArrayList vs Vector

```
===================
 These 2 classes along with Serializable,Cloneable,it also implements RandomAccess
 Any random elements present in ArrayList and Vector can be accesed through same
speed, becoz it
 is accessed using "RandomAccess".
 ArrayList and Vector is best suited when our frequent operation is read.

RandomAccess is a marker interface which is a part of java.util package,where the
required ability
is provided automatically by the jvm.

eg#1.
ArrayList l1= new ArrayList();
LinkedList l2=new LinkedList();
System.out.println(l1 instanceof Serializable);//true
System.out.println(l1 instanceof Cloneable);//true
System.out.println(l2 instanceof Serializable);//true
System.out.println(l2 instanceof Cloneable);//true
System.out.println(l1 instanceof RandomAccess);//true
System.out.println(l2 instanceof RandomAccess);//false

When to use ArrayList and when not to use?
 ArrayList => it is best suited if our frequent operation is "retrieval
operation",because
              it implements RandomAccess interface.

 ArrayList => it is the worst choice if our frequent operation is "insert/deletion"
in the middle
              because it should perform so many shift operations.To resolve this
problem we should
              use "LinkedList".


Differences b/w ArrayList and Vector?
 ArrayList => Most of the methods are not synchronized.
 Vector    => Most of the methods are synchronized.

 ArrayList =>It is not thread safe becoz mulitple threads can operate on a object.
 Vector    =>It is thread safe becoz only one thread is allowed to operate.

 ArrayList => performance is high becoz threads are not allowed to wait.
  Vector   => performance is relatively low becoz thread are required to wait.

 ArrayList => It is not a legacy class.
   Vector  => It is a legacy class.

How to use ArrayList, but thread safety is required how would u get or how to get
synchronized version of ArrayList?
  ArrayList l=new ArrayList();// now 'l' is nonsynchronized
  ArrayList l1=Collections.synchronizedList(l);// now 'l1' is synchronized

Note::These methods are a part of Collections class(utility class)
  public static List synchronizedList(List l);
  public static Map synchronizedMap(Map m);
  public static Set synchronizedSet(Set s);


eg#1.
import java.util.*;
import java.io.*;

public class Test
{
      public static void main(String[] args)
      {
            //List : index plays a role[internally Array]
            ArrayList al  = new ArrayList();
            LinkedList ll = new LinkedList();

            System.out.println("Arraylist implements Serializable :: "+(al
instanceof Serializable));
            System.out.println("Linkedlist implements Serializable ::"+(ll
instanceof Serializable));

            System.out.println();

            System.out.println("Arraylist implements Cloneable :: "+(al instanceof
Cloneable));
            System.out.println("Linkedlist implements Serializable ::"+(ll
instanceof Cloneable));

            System.out.println();
            System.out.println("Arraylist implements RandomAccess :: "+(al
instanceof RandomAccess));
            System.out.println("Linkedlist implements RandomeAccess ::"+(ll
instanceof RandomAccess));

      }
}
```

```
Output
Arraylist implements Serializable  :: true
Linkedlist implements Serializable :: true

Arraylist implements Cloneable     :: true
Linkedlist implements Serializable :: true

Arraylist implements RandomAccess  :: true
Linkedlist implements RandomeAccess:: false
```

LinkedList
==========

=> Memory management is done effectively if we work with LinkedList.
=> memory is not given in continous fashion.

a. DataStructure is :: doubly linked list
b. heterogenous objects are allowed
c. null insertion is possible
d. duplicates are allowed
e. linkedlist implements Serializable and Cloneable interface but not RandomAccess.

Usage
1. If our frequent operation is insertion/deletion in the middle then we need to opt for "LinkedList".

```
      LinkedList l=new LinkedList();
      l.add(a);
      l.add(10);
      l.add(z);
      l.add(2,'a');
      l.remove(3);
```
2. LinkedList is the worst choice if our frequent operation is retreival operation.

Constructors
============
 a. LinkedList l=new LinkedList();
       It creates a empty LinkedList object.

 b. LinkedList l=new LinkedList(Collection c);
      To convert any Collection object to LinkedList.

Methods associated with LinkedList
++++++++++++++++++++++++++++++++++
Normally we use LinkedList to implement stack and queue, to provide the support we use
stack -> push(),pop(),display()
queue -> insert(),delete(),display()

```
1. public E getFirst();
2. public E getLast();
3. public E removeFirst();
4. public E removeLast();
5. public void addFirst(E);
6. public void addLast(E);
```

eg#1.
```
import java.util.*;
public class Test
{
      public static void main(String[] args)
      {
            //Underlying datastructure :: doubly linked list
            LinkedList ll = new LinkedList();
            ll.add("pwskills");
            ll.add(30);
            ll.add("pwskills");
            ll.add(null);
            System.out.println(ll);

            System.out.println();
            ll.add(0,"nitin");

            System.out.println(ll);
            ll.set(0,"naveen");

            System.out.println(ll);
            ll.addFirst("dhoni");

            System.out.println(ll);
            ll.addLast("kohli");

            System.out.println(ll);
      }
}
```

```
Output
[pwskills, 30, pwskills, null]
[nitin, pwskills, 30, pwskills, null]
[naveen, pwskills, 30, pwskills, null]
[dhoni, naveen, pwskills, 30, pwskills, null]
[dhoni, naveen, pwskills, 30, pwskills, null, kohli]
```

Vector:
=> The Underlying Data Structure is Resizable Array ORGrowable Array.
=> Insertion Order is Preserved.
=> Duplicate Objects are allowed.
=> Heterogeneous Objects are allowed.
=> null Insertion is Possible.
=> Implements Serializable, Cloneable and RandomAccess interfaces.
=> Every Method Present Inside Vector is Synchronized and Hence Vector Object is Thread Safe.
=> Vector is the Best Choice if Our Frequent Operation is Retrieval.
=> Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle.

Constructors
============
1) Vector v = new Vector();
=> Creates an Empty Vector Object with Default Initial Capacity 10.
   Once Vector Reaches its Max Capacity then a New Vector Object will be Created with
      New Capacity = Current Capacity * 2

2) Vector v = new Vector(intinitialCapacity);
3) Vector v = new Vector(intinitialCapacity, intincrementalCapacity);
4) Vector v = new Vector(Collection c);

```
Methods:
1) To Add Elements:
      add(Object o)              -> Collection
      add(int index, Object o)   -> List
      addElement(Object o)       -> Vector

2) To Remove Elements:
      remove(Object o)           -> Collection
      removeElement(Object o)    -> Vector
      remove(int index)          -> List
      removeElementAt(int index) -> Vector
      clear()                    -> Collection
      removeAllElements()        -> Vector

3) To Retrive Elements:
      Object get(int index)       -> List
      Object elementAt(int index) -> Vector
      Object firstElement()       -> Vector
      Object lastElement()        -> Vector

4) Some Other Methods:
      int size()
      int capacity()
      Enumeration element()
```

eg#1.

```
import java.util.*;
public class Test
{
      public static void main(String[] args)
      {
            //Underlying datastructure :: Growable Array
            Vector v = new Vector();
            System.out.println(v.capacity());//10

            for (int i =1;i<=10 ;i++ )
            {
                  v.addElement(i);
            }
            System.out.println(v);
            System.out.println(v.capacity());

            v.addElement("sachin");
            v.addElement(null);

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

            v.removeElementAt(3);
            System.out.println(v);

            System.out.println();

            //Working with Cursor
            Enumeration e= v.elements();
            while (e.hasMoreElements())
            {
                  Object o =e.nextElement();
                  System.out.println(o);
            }

      }
}
```

Output

```
10
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
10
20
[1, 2, 3, 5, 6, 7, 8, 9, 10, sachin, null]

1
2
3
5
6
7
8
9
10
sachin
null
```