**Collections**

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

**Root Interface for Java Collection Classes:**

* Collection interface (java.util.Collection)
* Map interface (java.util.Map)

**Need for Collection Framework :**

* Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.
* Another drawback being that most of the ‘Vector’ methods are final, meaning we cannot extend the ’Vector’ class to implement a similar kind of Collection.
* Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced the Collection Framework in JDK 1.2.

**Advantages of Collection Framework:**

* **Consistent API:** The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
* **Reduces programming effort:** A programmer doesn’t have to worry about the design of Collection, and he can focus on its best use in his program.
* **Increases program speed and quality:** Increases performance by providing high-performance implementations of useful data structures and algorithms.

**Legacy Classes**

* Early version of java did not include the Collections framework. It only defined several classes and interfaces that provide methods for storing objects.
* When Collections framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes.
* All legacy classes and interface were redesign by JDK 5 to support Generics. In general, the legacy classes are supported because there is still some code that uses them.

The following are the legacy classes defined by java.util package

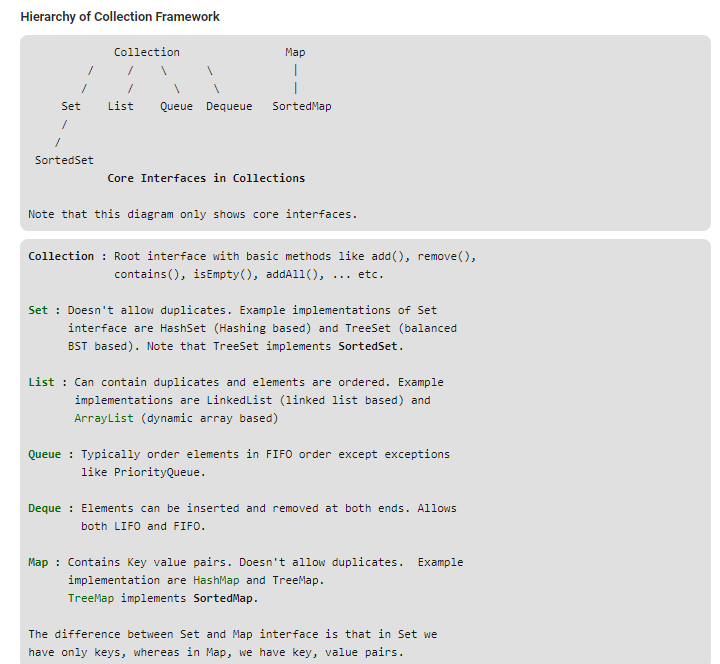
* Dictionary
* HashTable
* Properties
* Stack
* Vector

**\*\*There is only one legacy interface called Enumeration.**

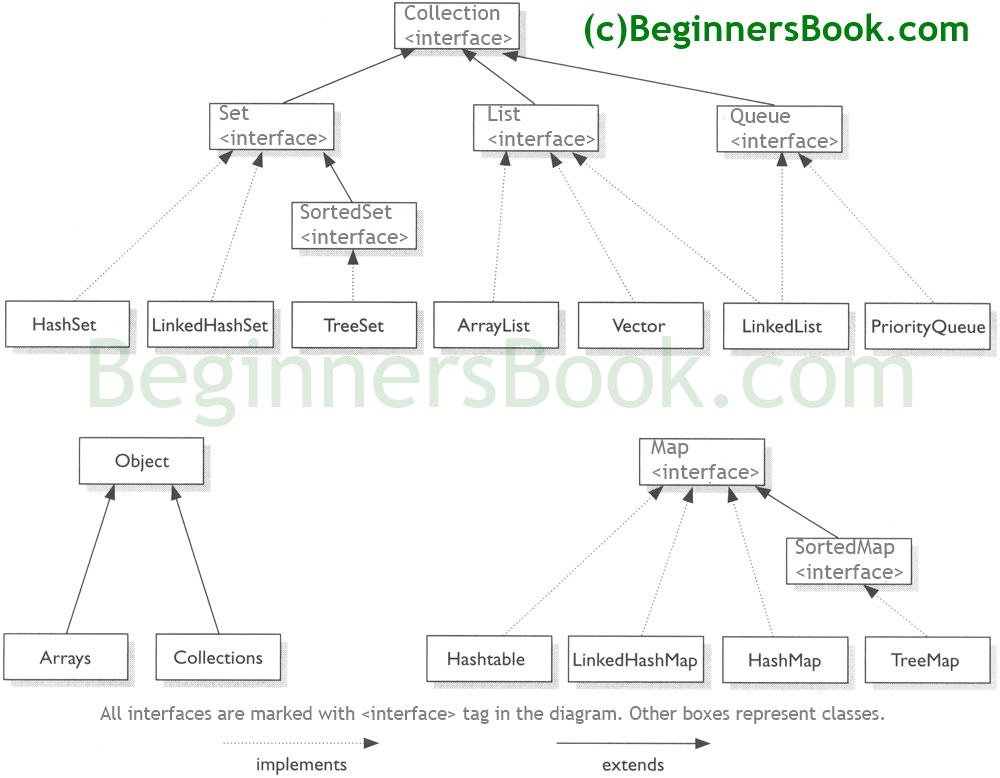
**NOTE**: All the legacy classes are synchronized.

**Enumeration interface**

* Enumeration interface defines method to enumerate(obtain one at a time) through collection of objects.
* This interface is superseded(replaced) by Iterator interface.
* However, some legacy classes such as Vector and Properties defines several method in which Enumeration interface is used.
* It specifies the following two methods
* boolean hasMoreElements() //It returns true while there are still more elements to extract, and returns false when all the elements have been enumerated.
* Object nextElement() //It returns the next object in the enumeration i.e. each call to nextElement() method obtains the next object in the enumeration. It throws NoSuchElementException when the enumeration is complete.



**Beginner’s Book:**

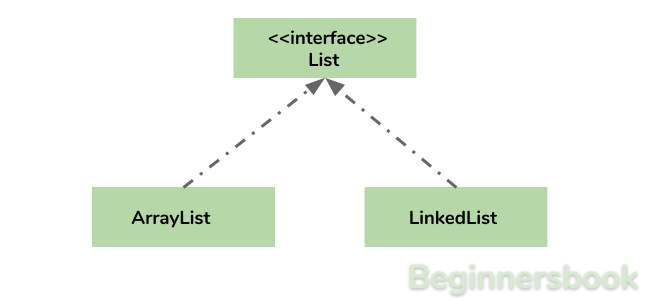


**List**

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.

**ArrayList**:

This class implements List interface and it is based on an Array data structure. ArrayList is a resizable-array implementation of the List interface. It implements all optional list operations, and permits all elements, including null.



**Why ArrayList is better than Array?**

The limitation with array is that it has a fixed length so if it is full you cannot add any more elements to it, likewise if there are number of elements gets removed from it the memory consumption would be the same as it doesn’t shrink.

On the other ArrayList can dynamically grow and shrink after addition and removal of elements (See the images below).

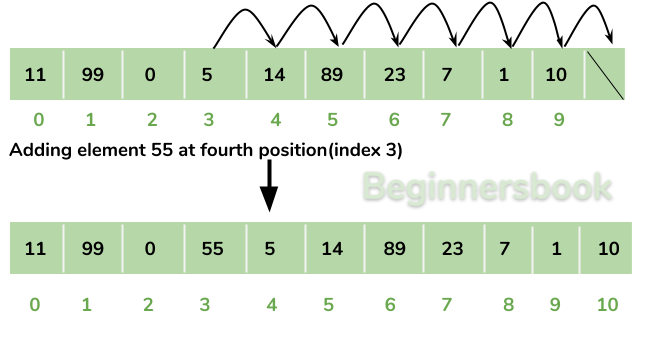
Apart from these benefits ArrayList class enables us to use predefined methods of it which makes our task easy.

ArrayList<String> alist=new ArrayList<String>();

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

alist.add("Steve"); //This will add "Steve" at the end of List

alist.add(3, "Steve"); //This will add "Steve" at the fourth position





## Methods of ArrayList class

1) **add( Object o)**: This method adds an object o to the arraylist.

obj.add("hello");

This statement would add a string hello in the arraylist at last position.

2) **add(int index, Object o)**: It adds the object o to the array list at the given index.

obj.add(2, "bye");

3) **remove(Object o)**: Removes the object o from the ArrayList.

obj.remove("Chaitanya");

4) **remove(int index)**: Removes element from a given index.

obj.remove(3);

5) **set(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.

obj.set(2, "Tom");

It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

6)**int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.

int pos = obj.indexOf("Tom");

7) **Object get(int index)**: It returns the object of list which is present at the specified index.

String str= obj.get(2);

Function get would return the string stored at 3rd position (index 2) and would be assigned to the string “str”. We have stored the returned value in string variable because in our example we have defined the ArrayList is of String type. If you are having integer array list then the returned value should be stored in an integer variable.

8) **int size()**: It gives the size of the ArrayList – Number of elements of the list.

int numberofitems = obj.size();

9) **boolean contains(Object o)**: It checks whether the given object o is present in the array list if its there then it returns true else it returns false.

obj.contains("Steve");

It would return true if the string “Steve” is present in the list else we would get false.

10) **clear():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

obj.clear();

### Method 1: Initialization using Arrays.asList

**Syntax:**

ArrayList<Type> obj = new ArrayList<Type>(

Arrays.asList(Object o1, Object o2, Object o3, ....so on));

**Example:**

import java.util.\*;

public class InitializationExample1 {

public static void main(String args[]) {

ArrayList<String> obj = new ArrayList<String>(

Arrays.asList("Pratap", "Peter", "Harsh"));

System.out.println("Elements are:"+obj);

}

}

Output:

Elements are:[Pratap, Peter, Harsh]

### Method 2: Anonymous inner class method to initialize ArrayList

**Syntax:**

ArrayList<T> obj = new ArrayList<T>(){{

add(Object o1);

add(Object o2);

add(Object o3);

...

...

}};

**Example:**

import java.util.\*;

public class InitializationExample2 {

public static void main(String args[]) {

ArrayList<String> cities = new ArrayList<String>(){{

add("Delhi");

add("Agra");

add("Chennai");

}};

System.out.println("Content of Array list cities:"+cities);

}

}

Output:

Content of Array list cities:[Delhi, Agra, Chennai]

### Method3: Normal way of ArrayList initialization

**Syntax:**

ArrayList<T> obj = new ArrayList<T>();

obj.add("Object o1");

obj.add("Object o2");

obj.add("Object o3");

...

...

**Example:**

import java.util.\*;

public class Details {

public static void main(String args[]) {

ArrayList<String> books = new ArrayList<String>();

books.add("Java Book1");

books.add("Java Book2");

books.add("Java Book3");

System.out.println("Books stored in array list are: "+books);

}

}

Output:

Books stored in array list are: [Java Book1, Java Book2, Java Book3]

### Method 4: Use Collections.ncopies

[Collections.ncopies](http://java.sun.com/javase/6/docs/api/java/util/Arrays.html#asList%28T...%29) method can be used when we need to initialize the ArrayList with the same value for all of its elements. **Syntax: count** is number of elements and **element** is the item value

ArrayList<T> obj = new ArrayList<T>(Collections.nCopies(count, element));

**Example:**

import java.util.\*;

public class Details {

public static void main(String args[]) {

ArrayList<Integer> intlist = new ArrayList<Integer>(Collections.nCopies(10, 5));

System.out.println("ArrayList items: "+intlist);

}

}

Output:

ArrayList items: [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]

**Looping in Arraylist:**

4 Ways:

1. For Loop
2. Advanced for loop
3. While Loop
4. Iterator
5. import java.util.\*;
6. public class LoopExample {
7. public static void main(String[] args) {
8. ArrayList<Integer> arrlist = new ArrayList<Integer>();
9. arrlist.add(14);
10. arrlist.add(7);
11. arrlist.add(39);
12. arrlist.add(40);
13. /\* For Loop for iterating ArrayList \*/
14. System.out.println("For Loop");
15. for (int counter = 0; counter < arrlist.size(); counter++) {
16. System.out.println(arrlist.get(counter));
17. }
18. /\* Advanced For Loop\*/
19. System.out.println("Advanced For Loop");
20. for (Integer num : arrlist) {
21. System.out.println(num);
22. }
23. /\* While Loop for iterating ArrayList\*/
24. System.out.println("While Loop");
25. int count = 0;
26. while (arrlist.size() > count) {
27. System.out.println(arrlist.get(count));
28. count++;
29. }
30. /\*Looping Array List using Iterator\*/
31. System.out.println("Iterator");
32. Iterator iter = arrlist.iterator();
33. while (iter.hasNext()) {
34. System.out.println(iter.next());
35. }
36. }
37. }

//using Enumeration interface

System.***out***.println("Enumeration");

Enumeration<Integer> en = Collections.*enumeration*(arrlist);

**while**(en.hasMoreElements()) {

System.***out***.println(en.nextElement());

}

**ArrayList Sorting :**

import java.util.\*;

public class Details {

public static void main(String args[]){

ArrayList<String> listofcountries = new ArrayList<String>();

listofcountries.add("India");

listofcountries.add("US");

listofcountries.add("China");

listofcountries.add("Denmark");

/\*Unsorted List\*/

System.out.println("Before Sorting:");

for(String counter: listofcountries){

System.out.println(counter);

}

/\* Sort statement\*/

Collections.sort(listofcountries);

/\* Sorted List\*/

System.out.println("After Sorting:");

for(String counter: listofcountries){

System.out.println(counter);

}

}

}

**Sorting in Descending order :**

We are using [Collections.reverseOrder()](https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html" \l "reverseOrder()" \t "_blank) method along with Collections.sort() in order to sort the list in decreasing order.

Collections.sort(arraylist, Collections.reverseOrder());

OR

Collections.sort(list);  
Collections.reverse(list);

import java.util.\*;

public class Details {

public static void main(String args[]){

ArrayList<String> arraylist = new ArrayList<String>();

arraylist.add("AA");

arraylist.add("ZZ");

arraylist.add("CC");

arraylist.add("FF");

/\*Unsorted List: ArrayList content before sorting\*/

System.out.println("Before Sorting:");

for(String str: arraylist){

System.out.println(str);

}

/\* Sorting in decreasing order\*/

Collections.sort(arraylist, Collections.reverseOrder());

/\* Sorted List in reverse order\*/

System.out.println("ArrayList in descending order:");

for(String str: arraylist){

System.out.println(str);

}

}

}

**Sorting using Comparable and Comparator:**

We generally use Collections.sort() method to sort a simple array list. However if the ArrayList is of **custom object** type then in such case you have two options for sorting- **comparable and comparator** interfaces.

While calling the Collections.sort() on the List of Objects and boom! I got the the error message like this –  
Exception in thread “main” java.lang.Error: Unresolved compilation problem:  
Bound mismatch: The generic method sort(List) of type Collections is not applicable for the arguments (ArrayList). The inferred type Student is not a valid substitute for the bounded parameter > at beginnersbook.com.Details.main(Details.java:11)

Reason: I Just called the sort method on an ArrayList of Objects which actually doesn’t work until unless we use interfaces like Comparable and Comparator.

**Comparable interface** is mainly used to sort the arrays (or lists) of **custom objects**.  
Lists (and arrays) of objects that implement Comparable interface can be sorted automatically by Collections.sort (and Arrays.sort).

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class Demo {

public static void main(String[] args) {

/\*

\* Integer class implements Comparable

\* Interface so we can use the sort method

\*/

int[] arr = {11,55,22,0,89};

Arrays.sort(arr);

System.out.print("Sorted Int Array: ");

System.out.println(Arrays.toString(arr));

/\*

\* String class implements Comparable

\* Interface so we can use the sort method

\*/

System.out.print("Sorted String Array: ");

String[] names = {"Steve", "Ajeet", "Kyle"};

Arrays.sort(names);

System.out.println(Arrays.toString(names));

/\*

\* String class implements Comparable

\* Interface so we can use the sort method

\*/

System.out.print("Sorted List: ");

List fruits = new ArrayList();

fruits.add("Orange");

fruits.add("Banana");

fruits.add("Apple");

fruits.add("Guava");

fruits.add("Grapes");

Collections.sort(fruits);

for(String s: fruits) System.out.print(s+", ");

}

}

It is to sort the Arrays and list of objects that implements Comparable interface, you just need to call the Collections.sort (and Arrays.sort).  
**However if you want to sort the objects of custom class then you need to implement the Comparable interface in our custom class.**

**Method Present:**

public abstract int compareTo(T obj)

Since this method is abstract, you must implement this method in your class if you implement the Comparable interface.

## Example: Sorting Custom object by implementing Comparable interface

As you can see I have implemented the Comparable interface in my Author class because I want to sort the objects of this class. I have written the logic of sorting in the compareTo() method, you can write logic based on the requirement. I wanted to sort the author names by last name first and if the last name is same then by first name. If you want to sort by the last name only then first line inside compareTo() method is enough.

**Author class**

public class Author implements Comparable<Author> {

String firstName;

String lastName;

String bookName;

Author(String first, String last, String book){

this.firstName = first;

this.lastName = last;

this.bookName = book;

}

@Override

/\*

\* This is where we write the logic to sort. This method sort

\* automatically by the first name in case that the last name is

\* the same.

\*/

public int compareTo(Author au){

/\*

\* Sorting by last name. compareTo should return < 0 if this(keyword)

\* is supposed to be less than au, > 0 if this is supposed to be

\* greater than object au and 0 if they are supposed to be equal.

\*/

int last = this.lastName.compareTo(au.lastName);

//Sorting by first name if last name is same d

return last == 0 ? this.firstName.compareTo(au.firstName) : last;

}

}

**Sorting class: SortAuthByNames**

import java.util.ArrayList;

import java.util.Collections;

public class SortAuthByNames{

public static void main(String args[]){

// List of objects of Author class

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

al.add(new Author("Henry","Miller", "Tropic of Cancer"));

al.add(new Author("Nalo","Hopkinson", "Brown Girl in the Ring"));

al.add(new Author("Frank","Miller", "300"));

al.add(new Author("Deborah","Hopkinson", "Sky Boys"));

al.add(new Author("George R. R.","Martin", "Song of Ice and Fire"));

/\*

\* Sorting the list using Collections.sort() method, we

\* can use this method because we have implemented the

\* Comparable interface in our user defined class Author

\*/

Collections.sort(al);

for(Author str:al){

System.out.println(str.firstName+" "+

str.lastName+" "+"Book: "+str.bookName);

}

}

}

**Output:**

Deborah Hopkinson Book: Sky Boys

Nalo Hopkinson Book: Brown Girl in the Ring

George R. R. Martin Book: A Song of Ice and Fire

Frank Miller Book: 300

Henry Miller Book: Tropic of Cancer

**Comparable did our job why do we need Comparator anymore?**  
Since Comparable is implemented by the same class whose objects are sorted so it binds you with that sorting logic which is ok in most of the cases but in case you want to have more than way of sorting your class objects you should use comparators.

**Note:**

All wrapper classes and String class implement Comparable interface. [Wrapper classes](https://howtodoinjava.com/java/basics/java-wrapper-classes/) are compared by their values, and strings are compared **lexicographically**.

#### 1.2**. Collections.sort() and Arrays.sort()**

1. Use Collections.sort() method sort a **list** of objects.
2. Use Arrays.sort() method sort an **array** of objects.

#### 1.3. **Collections.reverseOrder()**

This utility method returns a Comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface.

This enables a simple idiom for sorting (or maintaining) collections (or arrays) of objects that implement the Comparable interface in **reverse-natural-order**.

**Comparator :**

By using Comparable we can sort the objects based on any data member. If we want to sort the objects based on any of the data member then we can use Comparable but **what if we want to have multiple sort choices and we can sort objects based on any choice**, this can be done using Comparator interface, we can create as many Comparator as we want and then we can call Collections.sort on one or more Comparator.

//Sorting arraylist al by Author Age

Collections.sort(al, new AuthorAgeComparator());

//Sorting arraylist al by Book Name

Collections.sort(al, new BookNameComparator());

## Complete Comparator Example

**Author.java**

public class Author implements Comparable<Author> {

String firstName;

String bookName;

int auAge;

Author(String first, String book, int age){

this.firstName = first;

this.bookName = book;

this.auAge = age;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getBookName() {

return bookName;

}

public void setBookName(String bookName) {

this.bookName = bookName;

}

public int getAuAge() {

return auAge;

}

public void setAuAge(int auAge) {

this.auAge = auAge;

}

@Override

/\*

\* When we only use Comparable, this is where we write sorting

\* logic. This method is called when we implement the Comparable

\* interface in our class and call Collections.sort()

\*/

public int compareTo(Author au){

return this.firstName.compareTo(au.firstName);

}

}

**AuthorAgeComparator.java**

import java.util.\*;

class AuthorAgeComparator implements Comparator<Author>{

public int compare(Author a1,Author a2){

if(a1.auAge==a2.auAge)

return 0;

else if(a1.auAge>a2.auAge)

return 1;

else

return -1;

}

}

**BookNameComparator.java**

import java.util.\*;

public class BookNameComparator implements Comparator<Author>{

public int compare(Author a1,Author a2){

return a1.bookName.compareTo(a2.bookName);

}

}

**SortingPgm.java**

import java.util.ArrayList;

import java.util.Collections;

public class SortingPgm{

public static void main(String args[]){

// List of objects of Author class

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

al.add(new Author("Henry", "Tropic of Cancer",  45));

al.add(new Author("Nalo", "Brown Girl in the Ring", 56));

al.add(new Author("Frank", "300", 65));

al.add(new Author("Deborah", "Sky Boys", 51));

al.add(new Author("George R. R.", "A Song of Ice and Fire", 62));

/\*

\* Sorting the list using Collections.sort() method, we

\* can use this method because we have implemented the

\* Comparable interface in our user defined class Author

\*/

System.out.println("Sorting by Author First Name:");

Collections.sort(al);

for(Author au: al){

System.out.println(au.getFirstName()+", "+au.getBookName()+", "+

au.getAuAge());

}

/\*Sorting using AuthorAgeComparator\*/

System.out.println("Sorting by Author Age:");

Collections.sort(al, new AuthorAgeComparator());

for(Author au: al){

System.out.println(au.getFirstName()+", "+au.getBookName()+", "+

au.getAuAge());

}

/\*Sorting using BookNameComparator\*/

System.out.println("Sorting by Book Name:");

Collections.sort(al, new BookNameComparator());

for(Author au: al){

System.out.println(au.getFirstName()+", "+au.getBookName()+", "+

au.getAuAge());

}

}

}

## Getting sub-list from an ArrayList

The **subList method** returns a list therefore to store the sublist in another ArrayList we must need to type cast the returned value in same way as I did in the below example. On the other side if we are storing the returned sublist into a list then there is no need to type cast (Refer the example).

package beginnersbook.com;

import java.util.ArrayList;

import java.util.List;

public class SublistExample {

public static void main(String a[]){

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

//Addition of elements in ArrayList

al.add("Steve");

al.add("Justin");

al.add("Ajeet");

al.add("John");

al.add("Arnold");

al.add("Chaitanya");

System.out.println("Original ArrayList Content: "+al);

//Sublist to ArrayList

ArrayList<String> al2 = new ArrayList<String>(al.subList(1, 4));

System.out.println("SubList stored in ArrayList: "+al2);

//Sublist to List

List<String> list = al.subList(1, 4);

System.out.println("SubList stored in List: "+list);

}

}

# Synchronize ArrayList in java with example

As we are aware that ArrayList is non-synchronized and should not be used in multi-thread environment without explicit synchronization

**There are two ways to synchronize explicitly:**

1. Using Collections.synchronizedList() method
2. Using thread-safe variant of ArrayList: CopyOnWriteArrayList

### Example 1: Collections.synchronizedList() method for Synchronizing ArrayList

In this example we are using [Collections.synchronizedList()](https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html" \l "synchronizedList(java.util.List)" \t "_blank) method. The important point to note here is that iterator should be in synchronized block in this type of synchronization as shown in the below example.

package beginnersbook.com;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.Collections;

public class Details {

public static void main(String a[]){

List<String> syncal =

Collections.synchronizedList(new ArrayList<String>());

//Adding elements to synchronized ArrayList

syncal.add("Pen");

syncal.add("NoteBook");

syncal.add("Ink");

System.out.println("Iterating synchronized ArrayList:");

synchronized(syncal) {

Iterator<String> iterator = syncal.iterator();

while (iterator.hasNext())

System.out.println(iterator.next());

}

}

}

O/P

Iterating synchronized ArrayList:

Pen

NoteBook

Ink

### Method 2: Using CopyOnWriteArrayList

[CopyOnWriteArrayList](https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) is a thread-safe variant of ArrayList.

package beginnersbook.com;

import java.util.concurrent.CopyOnWriteArrayList;

import java.util.Iterator;

public class Details {

public static void main(String a[]){

CopyOnWriteArrayList<String> al = new CopyOnWriteArrayList<String>();

//Adding elements to synchronized ArrayList

al.add("Pen");

al.add("NoteBook");

al.add("Ink");

System.out.println("Displaying synchronized ArrayList Elements:");

//Synchronized block is not required in this method

Iterator<String> iterator = al.iterator();

while (iterator.hasNext())

System.out.println(iterator.next());

}

}

Output:

Displaying synchronized ArrayList Elements:

Pen

NoteBook

Ink

# Swap two elements in an ArrayList

public static void swap(List list, int i1, int i2)

This method swaps the element of index i1 with the element of index i2. It throws IndexOutOfBoundsException – if either i1 or i2 is less than zero or greater than the size of the list (i1 < 0 || i1 >= list.size() || i2 < 0 || i2 >= list.size()).

# Clone an ArrayList to another ArrayList

Object clone()

This method returns a shallow copy of the ArrayList instance.

## Complete example of ArrayList Cloning

In this example we have an ArrayList of String type and we are cloning it to another ArrayList using clone() method. The interesting point to see here is when we added and removed few elements from original ArrayList after the clone() method, the cloned ArrayList didn’t get affected. It shows that clone() method just returns a shallow copy of ArrayList.

package beginnersbook.com;

import java.util.ArrayList;

public class Details {

public static void main(String a[]){

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

//Adding elements to the ArrayList

al.add("Apple");

al.add("Orange");

al.add("Mango");

al.add("Grapes");

System.out.println("ArrayList: "+al);

ArrayList<String> al2 = (ArrayList<String>)al.clone();

System.out.println("Shallow copy of ArrayList: "+ al2);

//add and remove on original ArrayList

al.add("Fig");

al.remove("Orange");

//Display of both ArrayLists after add & remove

System.out.println("Original ArrayList:"+al);

System.out.println("Cloned ArrayList:"+al2);

}

}

Output:

ArrayList: [Apple, Orange, Mango, Grapes]

Shallow copy of ArrayList: [Apple, Orange, Mango, Grapes]

Original ArrayList:[Apple, Mango, Grapes, Fig]

Cloned ArrayList:[Apple, Orange, Mango, Grapes]

# Empty an ArrayList in Java

There are two ways to empty an ArrayList – By using [ArrayList.clear()](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html" \l "clear()" \t "_blank) method or with the help of [ArrayList.removeAll()](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html" \l "removeAll(java.util.Collection)" \t "_blank) method. Although both methods do the same task the way they empty the List is quite different.

The actual code of clear() method:

public void clear() {

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

arraylist[i] = null;

size = 0;

}

Here arraylist is an instance of ArrayList class.

Code of removeAll() method:

public boolean removeAll(Collection c) {

boolean ismodified = false;

Iterator iterator = iterator();

while (iterator.hasNext()) {

if (c.contains(iterator.next())) {

iterator.remove();

ismodified = true;

}

}

return ismodified;

}

By seeing the code of both the methods we can very well say that clear() method gives better performance compared to the removeAll() method.  
**Performance of clear**: O(n)  
**Performance of removeAll**: O(n^2)

# Java ArrayList trimToSize() Method example

trimToSize() method is used for memory optimization. It trims the capacity of ArrayList to the current list size. For e.g. An arraylist is having capacity of 15 but there are only 5 elements in it, calling trimToSize() method on this ArrayList would change the capacity from 15 to 5.

public void trimToSize()

package beginnersbook.com;

import java.util.ArrayList;

public class TrimExample {

public static void main(String args[]) {

ArrayList<Integer> arraylist = new ArrayList<Integer>(50);

arraylist.add(1);

arraylist.add(2);

arraylist.add(3);

arraylist.add(4);

arraylist.add(5);

arraylist.add(6);

arraylist.add(7);

arraylist.add(1);

arraylist.add(1);

arraylist.add(1);

arraylist.trimToSize();

System.out.println(arraylist);

}

}

[1, 2, 3, 4, 5, 6, 7, 1, 1, 1]

# Java ArrayList set() Method example

If there is a need to update the list element based on the index then set method of ArrayList class can be used. The method set(int index, Element E) updates the element of specified index with the given element E.

public E set(int index, Element E)

### Example:

In this example I have an ArrayList of Integer Type where I have added few elements and then I’m updating few of elements using set method of java.util.ArrayList class.

package beginnersbook.com;

import java.util.ArrayList;

public class SetExample {

public static void main(String args[]) {

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

arraylist.add(1);

arraylist.add(2);

arraylist.add(3);

arraylist.add(4);

arraylist.add(5);

arraylist.add(6);

arraylist.add(7);

System.out.println("ArrayList before update: "+arraylist);

//Updating 1st element

arraylist.set(0, 11);

//Updating 2nd element

arraylist.set(1, 22);

//Updating 3rd element

arraylist.set(2, 33);

//Updating 4th element

arraylist.set(3, 44);

//Updating 5th element

arraylist.set(4, 55);

System.out.println("ArrayList after Update: "+arraylist);

}

}

# Java ArrayList ensureCapacity() Method example

ArrayList internally implements growable dynamic array which means it can increase and decrease its size automatically. If we try to add an element to a already full ArrayList then it automatically re-sized internally to accommodate the new element however sometimes its not a good approach.

Consider a scenario when there is a need to add huge number of elements to an already full ArrayList, in such case ArrayList has to be resized several number of times which would result in a poor performance. For such scenarios ensureCapacity() method of Java.util.ArrayList class is very useful as it increases the size of the ArrayList by a specified capacity.

public void ensureCapacity(int minCapacity)

### Example

package beginnersbook.com;

import java.util.ArrayList;

public class EnsureCapacityExample {

public static void main(String args[]) {

// ArrayList with Capacity 4

ArrayList<String> al = new ArrayList<String>(4);

//Added 4 elements

al.add("Hi");

al.add("Hello");

al.add("Bye");

al.add("GM");

//Increase capacity to 5

al.ensureCapacity(5);

al.add("GE");

// let us print all the elements available in list

for (String temp: al) {

System.out.println(temp);

}

}

}

Hi

Hello

Bye

GM

GE

# Convert ArrayList to string array in java

In this post we have shared two methods of converting an [ArrayList](https://beginnersbook.com/2013/12/java-arraylist/" \o "ArrayList" \t "_blank)<String> to String array.

### Method 1: Manual way of conversion using ArrayList get() method

This is a manual way of copying all the ArrayList<String> elements to the String Array[]. In this example we have copied the whole list to array in three steps a) First we obtained the ArrayList size using size() method b) Fetched each element of the list using get() method and finally c) Assigned each element to corresponding array element using assignment = operator.

package beginnersbook.com;

import java.util.\*;

public class ArrayListTOArray {

public static void main(String[] args) {

/\*ArrayList declaration and initialization\*/

ArrayList<String> arrlist= new ArrayList<String>();

arrlist.add("String1");

arrlist.add("String2");

arrlist.add("String3");

arrlist.add("String4");

/\*ArrayList to Array Conversion \*/

String array[] = new String[arrlist.size()];

for(int j =0;j<arrlist.size();j++){

array[j] = arrlist.get(j);

}

/\*Displaying Array elements\*/

for(String k: array)

{

System.out.println(k);

}

}

}

**Output:**

String1

String2

String3

String4

### Method2: Conversion using toArray() method

In the above example we have manually copied each element of the array list to the array. However there is a method toArray() which can convert the ArrayList of string type to the array of Strings. More about [toArray() here](https://docs.oracle.com/javase/1.5.0/docs/api/java/util/ArrayList.html" \l "toArray()" \t "_blank).

package beginnersbook.com;

import java.util.\*;

public class Example {

public static void main(String[] args) {

/\*ArrayList declaration and initialization\*/

ArrayList<String> friendsnames= new ArrayList<String>();

friendsnames.add("Ankur");

friendsnames.add("Ajeet");

friendsnames.add("Harsh");

friendsnames.add("John");

/\*ArrayList to Array Conversion \*/

String frnames[]=friendsnames.toArray(new String[friendsnames.size()]);

/\*Displaying Array elements\*/

for(String k: frnames)

{

System.out.println(k);

}

}

}

Output:

Ankur

Ajeet

Harsh

John

# Convert an array to ArrayList in java

In the last tutorial we have shared two methods of [converting an ArrayList to Array with example](https://beginnersbook.com/2013/12/how-to-convert-arraylist-to-string-array-in-java/).

String array[] to ArrayList<String>

### Method 1: Conversion using Arrays.asList()

Syntax:

ArrayList<T> arraylist= new ArrayList<T>(Arrays.asList(arrayname));

Example:

In this example we are using [Arrays.asList](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html" \l "asList(T...)" \t "_blank) method to convert the Array to ArrayList.

import java.util.\*;

public class ArrayToArrayList {

public static void main(String[] args) {

/\* Array Declaration and initialization\*/

String citynames[]={"Agra", "Mysore", "Chandigarh", "Bhopal"};

/\*Array to ArrayList conversion\*/

ArrayList<String> citylist= new ArrayList<String>(Arrays.asList(citynames));

/\*Adding new elements to the converted List\*/

citylist.add("New City2");

citylist.add("New City3");

/\*Final ArrayList content display using for\*/

for (String str: citylist)

{

System.out.println(str);

}

}

}

Output:

Agra

Mysore

Chandigarh

Bhopal

New City2

New City3

### Method 2: Collections.addAll method

[Collections.addAll](https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#addAll(java.util.Collection,%20T...)) method all the array elements to the specified collection. This is how Collections.addAll method is being called. It does the same as Arrays.asList method however it is much faster than it so performance wise this is a best way to get the array converted to ArrayList.

String array[]={new Item(1), new Item(2), new Item(3), new Item(4)};  
ArrayList<T> arraylist = new ArrayList<T>();  
Collections.addAll(arraylist, array);

OR  
  
Collections.addAll(arraylist, new Item(1), new Item(2), new Item(3), new Item(4));  
  
Example

import java.util.\*;

public class Example2 {

public static void main(String[] args) {

/\* Array Declaration and initialization\*/

String array[]={"Hi", "Hello", "Howdy", "Bye"};

/\*ArrayList declaration\*/

ArrayList<String> arraylist= new ArrayList<String>();

/\*Conversion\*/

Collections.addAll(arraylist, array);

/\*Adding new elements to the converted List\*/

arraylist.add("String1");

arraylist.add("String2");

/\*Display array list\*/

for (String str: arraylist)

{

System.out.println(str);

}

}

}

Output

Hi

Hello

Howdy

Bye

String1

String2

### Method 3: Manual way of doing things

We can also add all the array’s element to the array list manually. Below example shows the logic of manual conversion.

package beginnersbook.com;

import java.util.\*;

public class Details {

public static void main(String[] args) {

/\*ArrayList declaration\*/

ArrayList<String> arraylist= new ArrayList<String>();

/\*Initialized Array\*/

String array[] = {"Text1","Text2","Text3","Text4"};

/\*array.length returns the current number of

\* elements present in array\*/

for(int i =0;i<array.length;i++)

{

/\* We are adding each array's element to the ArrayList\*/

arraylist.add(array[i]);

}

/\*ArrayList content\*/

for(String str: arraylist)

{

System.out.println(str);

}

}

}

Output:

Text1

Text2

Text3

Text4

# Difference between ArrayList and Vector In java

[ArrayList](https://beginnersbook.com/2013/12/java-arraylist/) and [Vector](https://beginnersbook.com/2013/12/vector-in-java/) both use Array as a data structure internally. However there are few differences in the way they store and process the data. In this post we will discuss the difference and similarities between ArrayList and Vector.

## ArrayList Vs Vector:

1) **Synchronization**: ArrayList is non-synchronized which means multiple threads can work on ArrayList at the same time. For e.g. if one thread is performing an add operation on ArrayList, there can be an another thread performing remove operation on ArrayList at the same time in a multithreaded environment

while Vector is synchronized. This means if one thread is working on Vector, no other thread can get a hold of it. Unlike ArrayList, only one thread can perform an operation on vector at a time.

2) **Resize:** Both ArrayList and Vector can grow and shrink dynamically to maintain the optimal use of storage, however the way they resized is different. ArrayList grow by half of its size when resized while Vector doubles the size of itself by default when grows.

3) **Performance**: ArrayList gives better performance as it is non-synchronized. Vector operations gives poor performance as they are thread-safe, the thread which works on Vector gets a lock on it which makes other thread wait till the lock is released.

4) **fail-fast**: First let me explain what is fail-fast: If the collection (ArrayList, vector etc) gets structurally modified by any means, except the **add or remove methods** of iterator, after creation of iterator then the iterator will throw [ConcurrentModificationException](https://docs.oracle.com/javase/6/docs/api/java/util/ConcurrentModificationException.html" \t "_blank). Structural modification refers to the addition or deletion of elements from the collection.

As per the [Vector javadoc](https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html) the Enumeration returned by Vector is not fail-fast. On the other side the iterator and listIterator returned by ArrayList are fail-fast.

5) **Who belongs to collection framework really?**The vector was not the part of collection framework, it has been included in collections later. It can be considered as Legacy code. There is nothing about Vector which List collection cannot do. Therefore Vector should be avoided. If there is a need of thread-safe operation make ArrayList synchronized as discussed in the next section of this post or use [CopyOnWriteArrayList](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html" \t "_blank) which is a thread-safe variant of ArrayList.

There are few **similarities between** these classes which are as follows:

1. Both Vector and ArrayList use growable array data structure.
2. The iterator and listIterator returned by these classes (Vector and ArrayList) are fail-fast.
3. They both are ordered collection classes as they maintain the elements insertion order.
4. Vector & ArrayList both allows duplicate and null values.
5. They both grows and shrinks automatically when overflow and deletion happens.

### When to use ArrayList and when to use vector?

It totally depends on the requirement. If there is a need to perform “thread-safe” operation the vector is your best bet as it ensures that only one thread access the collection at a time.

**Performance:** Synchronized operations consumes more time compared to non-synchronized ones so if there is no need for thread safe operation, ArrayList is a better choice as performance will be improved because of the concurrent processes.

**How to make ArrayList synchronized?**  
As I stated above ArrayList methods are non-synchronized but still if there is a need you can make them synchronized like this –

//Use Collecions.synzhonizedList method

List list = Collections.synchronizedList(new ArrayList());

...

//If you wanna use iterator on the synchronized list, use it

//like this. It should be in synchronized block.

synchronized (list) {

Iterator iterator = list.iterator();

while (iterator.hasNext())

...

iterator.next();

...

}

# Difference between ArrayList and HashMap in Java

[ArrayList](https://beginnersbook.com/2013/12/java-arraylist/) and [HashMap](https://beginnersbook.com/2013/12/hashmap-in-java-with-example/) are two commonly used collection classes in Java. Even though both are the part of collection framework, the way they store and process the data is entirely different. In this post we will see the main differences between these two collections.

## ArrayList vs HashMap in Java

1) **Implementation**: **ArrayList** implements List Interface while **HashMap** is an implementation of Map interface. List and Map are two entirely different collection interfaces.

2) **Memory consumption**: ArrayList stores the element’s value alone and internally maintains the indexes for each element.

ArrayList<String> arraylist = new ArrayList<String>();

//String value is stored in array list

arraylist.add("Test String");

HashMap stores key & value pair. For each value there must be a key associated in HashMap. That clearly shows that memory consumption is high in HashMap compared to the ArrayList.

HashMap<Integer, String> hmap= new HashMap<Integer, String>();

//String value stored along with the key value in hash map

hmap.put(123, "Test String");

3) **Order**: ArrayList maintains the insertion order while HashMap doesn’t. Which means ArrayList returns the list items in the same order in which they got inserted into the list. On the other side HashMap doesn’t maintain any order, the returned key-values pairs are not sorted in any kind of order.

4) **Duplicates**: ArrayList allows duplicate elements but HashMap doesn’t allow duplicate keys (It does allow duplicate values).

5) **Nulls**: ArrayList can have any number of null elements. HashMap allows one null key and any number of null values.

6) **get method**: In ArrayList we can **get** the element by specifying the index of it. In HashMap the elements is being fetched by specifying the corresponding key.

# Difference between ArrayList and LinkedList in Java

[ArrayList](https://beginnersbook.com/2013/12/java-arraylist/) and [LinkedList](https://beginnersbook.com/2013/12/linkedlist-in-java-with-example/) both implements List interface and their methods and results are almost identical. However there are few differences between them which make one better over another depending on the requirement.

## ArrayList Vs LinkedList

1) **Search**: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements **doubly linked list** which requires the traversal through all the elements for searching an element.

2) **Deletion**: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) **Inserts Performance**: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

4) **Memory Overhead**: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

There are few **similarities between** these classes which are as follows:

1. Both ArrayList and LinkedList are implementation of List interface.
2. They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.
3. Both these classes are non-synchronized and can be made synchronized explicitly by using [Collections.synchronizedList](https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html" \l "synchronizedList(java.util.List)" \t "_blank) method.
4. The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator’s own remove or add methods, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/6/docs/api/java/util/ConcurrentModificationException.html" \t "_blank)).

### When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

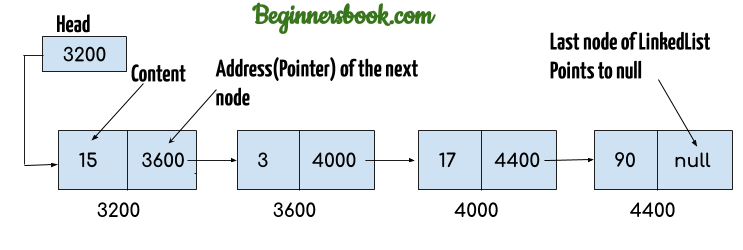
# **LinkedList in Java**

**LinkedList is a linear data structure**. However LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.

Each element in the LinkedList is called the **Node**. Each Node of the LinkedList contains two items:

1) Content of the element

2) Pointer/Address/Reference to the Next Node in the LinkedList.



**Note:**  
1. **Head** of the LinkedList only contains the Address of the **First element** of the List.  
2. The Last element of the LinkedList contains **null** in the pointer part of the node because it is the end of the List so it doesn’t point to anything as shown in the above diagram.  
3. The diagram which is shown above represents a **singly linked list**. There is another complex type variation of LinkedList which is called **doubly linked list**, node of a doubly linked list contains three parts:

1) Pointer to the previous node of the linked list

2) content of the element

3) pointer to the next node of the linked list.

## Why do we need a Linked List?

You must be aware of the arrays which is also a linear data structure but **arrays have certain limitations such as:**

1) **Size of the array is fixed** which is decided when we create an array so it is hard to predict the number of elements in advance, if the declared size fall short then we cannot increase the size of an array and if we declare a large size array and do not need to store that many elements then it is a waste of memory.

2) Array elements **need contiguous memory locations** to store their values.

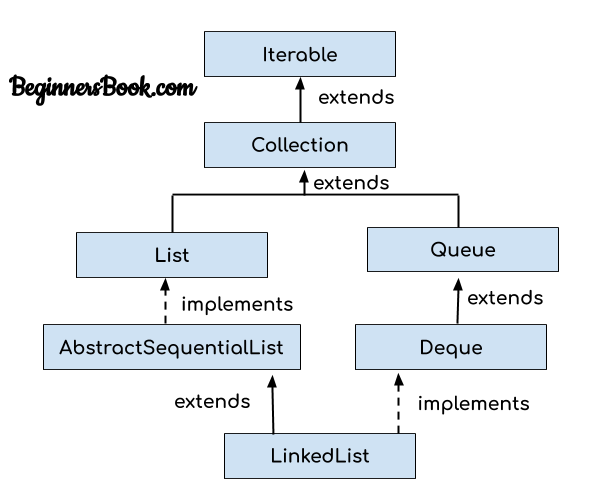
3) **Inserting an element in an array is performance wise expensive** as we have to shift several elements to make a space for the new element.

Similarly **deleting an element** from the array is also a performance wise expensive operation because all the elements after the deleted element have to be shifted left.

**These limitations are handled in the Linked List by providing following features:**  
1. Linked list allows **dynamic memory allocation**, which means memory allocation is done at the run time by the compiler and we do not need to mention the size of the list during linked list declaration.

2. Linked list elements **don’t need contiguous memory locations** because elements are linked with each other using the reference part of the node that contains the address of the next node of the list.

3. Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list doesn’t require element shifting, only the pointer of the previous and the next node requires change.



**Example:**

package com.beginnersbook;

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

LinkedList<String> list=new LinkedList<String>();

//Adding elements to the Linked list

list.add("Steve");

list.add("Carl");

list.add("Raj");

//Adding an element to the first position

list.addFirst("Negan");

//Adding an element to the last position

list.addLast("Rick");

//Adding an element to the 3rd position

list.add(2, "Glenn");

//Iterating LinkedList

Iterator<String> iterator=list.iterator();

while(iterator.hasNext()){

System.out.println(iterator.next());

}

}

}

## Methods of LinkedList class:

For all the examples in the below methods, consider llistobj as a reference for LinkedList<String>.

LinkedList<String> llistobj  = new LinkedList<String>();

1) **boolean add(Object item)**: It adds the item at the end of the list.

llistobj.add("Hello");

It would add the string “Hello” at the end of the linked list.

2) **void add(int index, Object item)**: It adds an item at the given index of the the list.

llistobj.add(2, "bye");

This will add the string “bye” at the 3rd position( 2 index is 3rd position as index starts with 0).

3) **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null. Consider the below example –

LinkedList<String> llistobj = new LinkedList<String>();

ArrayList<String> arraylist= new ArrayList<String>();

arraylist.add("String1");

arraylist.add("String2");

llistobj.addAll(arraylist);

This piece of code would add all the elements of ArrayList to the LinkedList.

4) **boolean addAll(int index, Collection c)**: It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.

llistobj.add(5, arraylist);

It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).

5) **void addFirst(Object item)**: It adds the item (or element) at the first position in the list.

llistobj.addFirst("text");

It would add the string “text” at the beginning of the list.

6) **void addLast(Object item)**: It inserts the specified item at the end of the list.

llistobj.addLast("Chaitanya");

This statement will add a string “Chaitanya” at the end position of the linked list.

7) **void clear()**: It removes all the elements of a list.

llistobj.clear();

8) **Object clone()**: It returns the copy of the list.

For e.g. My linkedList has four items: text1, text2, text3 and text4.

Object str= llistobj.clone();

System.out.println(str);

Output: The output of above code would be:

[text1, text2, text3, text4]

9) **boolean contains(Object item)**: It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

boolean var = llistobj.contains("TestString");

It will check whether the string “TestString” exist in the list or not.

10) **Object get(int index)**: It returns the item of the specified index from the list.

Object var = llistobj.get(2);

It will fetch the 3rd item from the list.

11) **Object getFirst()**: It fetches the first item from the list.

Object var = llistobj.getFirst();

12) **Object getLast()**: It fetches the last item from the list.

Object var= llistobj.getLast();

13) **int indexOf(Object item)**: It returns the index of the specified item.

llistobj.indexOf("bye");

14) **int lastIndexOf(Object item)**: It returns the index of last occurrence of the specified element.

int pos = llistobj.lastIndexOf("hello);

integer variable pos will be having the index of last occurrence of string “hello”.

15) **Object poll()**: It returns and removes the first item of the list.

Object o = llistobj.poll();

16) **Object pollFirst()**: same as poll() method. Removes the first item of the list.

Object o = llistobj.pollFirst();

17) **Object pollLast()**: It returns and removes the last element of the list.

Object o = llistobj.pollLast();

18) **Object remove()**: It removes the first element of the list.

llistobj.remove();

19) **Object remove(int index)**: It removes the item from the list which is present at the specified index.

llistobj.remove(4);

It will remove the 5th element from the list.

20) **Object remove(Object obj)**: It removes the specified object from the list.

llistobj.remove("Test Item");

21) **Object removeFirst()**: It removes the first item from the list.

llistobj.removeFirst();

22) **Object removeLast()**: It removes the last item of the list.

llistobj.removeLast();

23) **Object removeFirstOccurrence(Object item)**: It removes the first occurrence of the specified item.

llistobj.removeFirstOccurrence("text");

It will remove the first occurrence of the string “text” from the list.

24) **Object removeLastOccurrence(Object item)**: It removes the last occurrence of the given element.

llistobj.removeLastOccurrence("String1);

It will remove the last occurrence of string “String1”.

25) **Object set(int index, Object item)**: It updates the item of specified index with the give value.

llistobj.set(2, "Test");

It will update the 3rd element with the string “Test”.

26)**int size()**: It returns the number of elements of the list.

llistobj.size();

**Loop LinkedList**

1. For loop
2. Advanced For loop
3. Iterator
4. While Loop
5. package beginnersbook.com;
6. import java.util.\*;
7. public class LinkedListExample {
8. public static void main(String args[]) {
9. /\*LinkedList declaration\*/
10. LinkedList<String> linkedlist=new LinkedList<String>();
11. linkedlist.add("Apple");
12. linkedlist.add("Orange");
13. linkedlist.add("Mango");
14. /\*for loop\*/
15. System.out.println("\*\*For loop\*\*");
16. for(int num=0; num<linkedlist.size(); num++)
17. {
18. System.out.println(linkedlist.get(num));
19. }
20. /\*Advanced for loop\*/
21. System.out.println("\*\*Advanced For loop\*\*");
22. for(String str: linkedlist)
23. {
24. System.out.println(str);
25. }
26. /\*Using Iterator\*/
27. System.out.println("\*\*Iterator\*\*");
28. Iterator i = linkedlist.iterator();
29. while (i.hasNext()) {
30. System.out.println(i.next());
31. }
32. /\* Using While Loop\*/
33. System.out.println("\*\*While Loop\*\*");
34. int num = 0;
35. while (linkedlist.size() > num) {
36. System.out.println(linkedlist.get(num));
37. num++;
38. }
39. }
40. }

**Note:**

public boolean remove(Object o): Removes the first occurrence of the specified element from this list, if it is present. If this list does not contain the element, it is unchanged. Returns true if this list contained the specified element (or equivalently, if this list changed as a result of the call).

# **Get first and last elements from LinkedList**

1) public E getFirst(): Returns the first element in this list.  
2) public E getLast(): Returns the last element in this list.

**package** linkedlist\_examples;

**import** java.util.LinkedList;

**public** **class** LinkedListGet {

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

// Create a LinkedList

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

// Add elements to LinkedList

linkedlist.add("Item1");

linkedlist.add("Item2");

linkedlist.add("Item3");

linkedlist.add("Item4");

linkedlist.add("Item5");

linkedlist.add("Item6");

// Getting First element of the List

Object firstElement = linkedlist.getFirst();

System.***out***.println("First Element is: " + firstElement);

// Getting Last element of the List

Object lastElement = linkedlist.getLast();

System.***out***.println("Last Element is: " + lastElement);

// Getting all elements

**for**(**int** i=0; i < linkedlist.size(); i++){

System.***out***.println("Element at index "+i+" is: "+linkedlist.get(i));

}

}

}

**Note:**

public int indexOf(Object o): Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

public int lastIndexOf(Object o): Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

subList(int startIndex, int endIndex) method of LinkedList class. It returns a List between the specified index startIndex(inclusive) and endIndex(exclusive). Any changes made to the sublist will be reflected in the original list

# **LinkedList ListIterator**

Using Listterator we can iterate the list in both the directions(forward and backward). Along with traversing, we can also modify the list during iteration, and obtain the iterator’s current position in the list.

import java.util.LinkedList;

import java.util.ListIterator;

public class ListIteratorExample {

public static void main(String[] args) {

// Create a LinkedList

LinkedList<String> linkedlist = new LinkedList<String>();

// Add elements to LinkedList

linkedlist.add("Delhi");

linkedlist.add("Agra");

linkedlist.add("Mysore");

linkedlist.add("Chennai");

linkedlist.add("Pune");

// Obtaining ListIterator

ListIterator listIt = linkedlist.listIterator();

// Iterating the list in forward direction

System.out.println("Forward iteration:");

while(listIt.hasNext()){

System.out.println(listIt.next());

}

// Iterating the list in backward direction

System.out.println("\nBackward iteration:");

while(listIt.hasPrevious()){

System.out.println(listIt.previous());

}

}

}

**For iterating in reverse order:**

Iterator it = list.descendingIterator();

# **Clone a generic LinkedList in Java**

import java.util.LinkedList;

class LinkedListClone {

public static void main(String[] args) {

// create a LinkedList

LinkedList<String> list = new LinkedList<String>();

// Adding elements to the LinkedList

list.add("Element1");

list.add("Element2");

list.add("Element3");

list.add("Element4");

// Displaying LinkedList elements

System.out.println("LinkedList elements: "+list);

// Creating another list

LinkedList<String> list2 = new LinkedList<String>();

// Clone list to list2

/\* public Object clone(): Returns a shallow copy of this

\* LinkedList. (The elements themselves are not cloned.)

\*/

list2 = (LinkedList)list.clone();

// Displaying elements of second LinkedList

System.out.println("List 2 Elements: "+list2);

}

}

**Output:**

LinkedList elements: [Element1, Element2, Element3, Element4]

List 2 Elements: [Element1, Element2, Element3, Element4]

# **LinkedList push() and pop() methods**

public void push(E e): Inserts the element at the front of the list.

public E pop(): Removes and returns the first element of the list.

import java.util.LinkedList;

class LinkedListExample {

public static void main(String[] args) {

// Create a LinkedList of Strings

LinkedList<String> list = new LinkedList<String>();

// Add few Elements

list.add("Jack");

list.add("Robert");

list.add("Chaitanya");

list.add("kate");

// Display LinkList elements

System.out.println("LinkedList contains: "+list);

// push Element the list

list.push("NEW ELEMENT");

// Display after push operation

System.out.println("LinkedList contains: "+list);

}

}

**Note:**

#### LinkedList.poll()

Retrieves and removes the head (first element) of this list.

#### LinkedList.pollFirst()

public E pollFirst(): Retrieves and removes the first element of this list, or returns null if this list is empty.

#### LinkedList.pollLast()

public E pollLast(): Retrieves and removes the last element of this list, or returns null if this list is empty.

import java.util.LinkedList;

class LinkedListPollMethod{

public static void main(String[] args) {

// Create a LinkedList of Strings

LinkedList<String> list = new LinkedList<String>();

// Add few Elements

list.add("Element1");

list.add("Element2");

list.add("Element3");

list.add("Element4");

// Display LinkList elements

System.out.println("LinkedList before: "+list);

/\* poll(): Retrieves and removes the head (first element)

\* of this list.

\*/

System.out.println("Element removed: "+list.poll());

// Displaying list elements after poll() operation

System.out.println("LinkedList after: "+list);

}

}

public E peek(): Retrieves, but does not remove, the head (first element) of this list.

public E peekFirst(): Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.

public E peekLast(): Retrieves, but does not remove, the last element of this list, or returns null if this list is empty.

import java.util.LinkedList;

class LinkedListPeekDemo{

public static void main(String[] args) {

// Create a LinkedList of Strings

LinkedList<String> list = new LinkedList<String>();

// Add few Elements

list.add("Element1");

list.add("Element2");

list.add("Element3");

list.add("Element4");

// Display LinkList elements

System.out.println("LinkedList before: "+list);

//peek()

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

//peekFirst()

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

//peekLast()

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

// Should be same as peek methods does not remove

System.out.println("LinkedList after: "+list);

}

}

Ouput

LinkedList before: [Element1, Element2, Element3, Element4]

Element1

Element1

Element4

LinkedList after: [Element1, Element2, Element3, Element4]

# **Convert a LinkedList to ArrayList**

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.List;

public class ConvertExample {

public static void main(String[] args) {

LinkedList<String> linkedlist = new LinkedList<String>();

linkedlist.add("Harry");

linkedlist.add("Jack");

linkedlist.add("Tim");

linkedlist.add("Rick");

linkedlist.add("Rock");

List<String> list = new ArrayList<String>(linkedlist);

for (String str : list){

System.out.println(str);

}

}

}

# **LinkedList to array using toArray()**

import java.util.LinkedList;

public class ConvertExample {

public static void main(String[] args) {

//Creating and populating LinkedList

LinkedList<String> linkedlist = new LinkedList<String>();

linkedlist.add("Harry");

linkedlist.add("Maddy");

linkedlist.add("Chetan");

linkedlist.add("Chauhan");

linkedlist.add("Singh");

//Converting LinkedList to Array

String[] array = linkedlist.toArray(new String[linkedlist.size()]);

//Displaying Array content

System.out.println("Array Elements:");

for (int i = 0; i < array.length; i++)

{

System.out.println(array[i]);

}

}

}

## ArrayList Vs LinkedList

1) **Search**: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements **doubly linked list** which requires the traversal through all the elements for searching an element.

2) **Deletion**: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) **Inserts Performance**: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

4) **Memory Overhead**: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

There are few **similarities between** these classes which are as follows:

1. Both ArrayList and LinkedList are implementation of List interface.
2. They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.
3. Both these classes are non-synchronized and can be made synchronized explicitly by using [Collections.synchronizedList](https://docs.oracle.com/javase/6/docs/api/java/util/Collections.html" \l "synchronizedList(java.util.List)" \t "_blank) method.
4. The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator’s own remove or add methods, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/6/docs/api/java/util/ConcurrentModificationException.html" \t "_blank)).

### When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

**Vector:**

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

#### Three ways to create vector class object:

**Method 1:**

Vector vec = new Vector();

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

**Method 2:**  
Syntax: Vector object= new Vector(int initialCapacity)

Vector vec = new Vector(3);

It will create a Vector of initial capacity of 3.

**Method 3:**  
Syntax:

Vector object= new vector(int initialcapacity, capacityIncrement)

Example:

Vector vec= new Vector(4, 6)

Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

***Methods:***

1. **void addElement(Object element):** It inserts the element at the end of the Vector.
2. **int capacity():** This method returns the current capacity of the vector.
3. **int size():** It returns the current size of the vector.
4. **void setSize(int size):** It changes the existing size with the specified size.
5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement():** It is used for getting the first element of the vector.
9. **Object lastElement():** Returns the last element of the array.
10. **Object get(int index):** Returns the element at the specified index.
11. **boolean isEmpty():** This method returns true if Vector doesn’t have any element.
12. **boolean removeElement(Object element):** Removes the specifed element from vector.
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.

**Sublist in vector:**

public List subList(int fromIndex, int toIndex):

It returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive. (If fromIndex and toIndex are equal, the returned List is empty.) The returned List is backed by this List, so changes in the returned List are reflected in this List, and vice-versa. The returned List supports all of the optional List operations supported by this List.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a List can be used as a range operation by operating on a subList view instead of a whole List. For example, the following idiom removes a range of elements from a List:  
list.subList(from, to).clear();

# **Sort Vector using Collections.sort in java – Example**

[Vector](https://beginnersbook.com/2013/12/vector-in-java/) maintains the insertion order which means it displays the elements in the same order, in which they got added to the Vector. In this example, we will see how to sort Vector elements in ascending order by using [Collections.sort()](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List)).

import java.util.Collections;

import java.util.Vector;

public class SortingVectorExample {

public static void main(String[] args) {

// Create a Vector

Vector<String> vector = new Vector<String>();

//Add elements to Vector

vector.add("Walter");

vector.add("Anna");

vector.add("Hank");

vector.add("Flynn");

vector.add("Tom");

// By Default Vector maintains the insertion order

System.out.println("Vector elements before sorting: ");

for(int i=0; i < vector.size(); i++){

//get(i) method fetches the element from index i

System.out.println(vector.get(i));

}

// Collection.sort() sorts the collection in ascending order

Collections.sort(vector);

//Display Vector elements after sorting using Collection.sort

System.out.println("Vector elements after sorting: :");

for(int i=0; i < vector.size(); i++){

System.out.println(vector.get(i));

}

}

}

**Output:**

Vector elements before sorting:

Walter

Anna

Hank

Flynn

Tom

Vector elements after sorting: :

Anna

Flynn

Hank

Tom

Walter

# **Search elements in Vector using index**

In this tutorial, we will learn four following ways to search elements in Vector using index value.

1) public int indexOf(Object o): It returns the index of first occurrence of Object o in Vector.  
2) public int indexOf(Object o, int startIndex): It returns the index of the first occurrence of the Object o in this vector, searching forwards from startIndex (inclusive).  
3) public int lastIndexOf(Object o): It returns the index of last occurrence of Object o in Vector.  
4) public int lastIndexOf(Object o, int startIndex): It returns the index of the last occurrence of the specified element in this vector, searching backwards from startIndex(inclusive).

#### Example

import java.util.Vector;

public class SearchVector {

public static void main(String[] args) {

// Create a Vector object

Vector<String> vector = new Vector<String>();

//Add elements to Vector

vector.add("Kate");

vector.add("Patt");

vector.add("Kluge");

vector.add("Karon");

vector.add("Patt");

vector.add("Monica");

vector.add("Patt");

//This would return the index of first occurrence

int first\_index = vector.indexOf("Patt");

System.out.println("First Occurrence of Patt at index: "+first\_index);

//This would return the index of last occurrence

int last\_index = vector.lastIndexOf("Patt");

System.out.println("Last Occurrence of Patt at index: "+last\_index);

//This would start search from index 2(inclusive)

int after\_index = vector.indexOf("Patt", 2);

System.out.println("Occurrence after index 2: "+after\_index);

//This would search the element backward starting from index 6(inclusive)

int before\_index = vector.lastIndexOf("Patt", 6);

System.out.println("Occurrence before index 6: "+before\_index);

}

}

**Output:**

First Occurrence of Patt at index: 1

Last Occurrence of Patt at index: 6

Occurrence after index 2: 4

Occurrence before index 6: 6

**Note:**

[Collections.copy()](https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#copy(java.util.List,%20java.util.List)) method: Copy one vector to vector.

public E remove(int index): Removes the element at the specified position in this Vector. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the [Vector](https://beginnersbook.com/2013/12/vector-in-java/).

public void clear(): Removes all of the elements from this Vector. The Vector will be empty after this method call.

public E set(int index, E element): Replaces the element at the specified position in this Vector with the specified element.

**Vector Size :**

We can set the size of a Vector using setSize() method of [Vector class](https://beginnersbook.com/2013/12/vector-in-java/). If new size is greater than the current size then all the elements after current size index have null values. If new size is less than current size then the elements after current size index have been deleted from the Vector.

import java.util.Vector;

public class SetSizeExample {

public static void main(String[] args) {

// Create a Vector

Vector<String> vector = new Vector<String>();

//Add elements to Vector

vector.add("Walter");

vector.add("Anna");

vector.add("Hank");

vector.add("Flynn");

vector.add("Tom");

//Setting up the size greater than current size

vector.setSize(10);

System.out.println("Vector size: "+vector.size());

System.out.println("Vector elements: ");

for(int i=0; i < vector.size(); i++){

//get(i) method fetches the element from index i

System.out.println(vector.get(i));

}

//Setting up the size less than current size

vector.setSize(4);

System.out.println("\nVector size: "+vector.size());

System.out.println("Vector elements: ");

for(int i=0; i < vector.size(); i++){

System.out.println(vector.get(i));

}

}

}

**Output:**

Vector size: 10

Vector elements:

Walter

Anna

Hank

Flynn

Tom

null

null

null

null

null

Vector size: 4

Vector elements:

Walter

Anna

Hank

Flynn

# **Convert Vector to List**

Collections.list(vector.elements()) which returns a List object.

import java.util.Vector;

import java.util.List;

import java.util.Collections;

public class VectorToList {

public static void main(String[] args) {

// Step1: Creating a Vector of String elements

Vector<String> vector = new Vector<String>();

// Step2: Populating Vector

vector.add("Tim");

vector.add("Rock");

vector.add("Hulk");

vector.add("Rick");

vector.add("James");

// Step3: Displaying Vector elements

System.out.println("Vector Elements :");

for (String str : vector){

System.out.println(str);

}

// Step4: Converting Vector to List

List<String> list = Collections.list(vector.elements());

// Step 5: Displaying List Elements

System.out.println("\nList Elements :");

for (String str2 : list){

System.out.println(str2);

}

}

}

**Output:**

Vector Elements :

Tim

Rock

Hulk

Rick

James

List Elements :

Tim

Rock

Hulk

Rick

James

**Vector to ArrayList:**

ArrayList list = new ArrayList(vector);

**Vector to StringArray:**

public String toString(): It returns a string representation of this Vector, containing the String representation of each element.