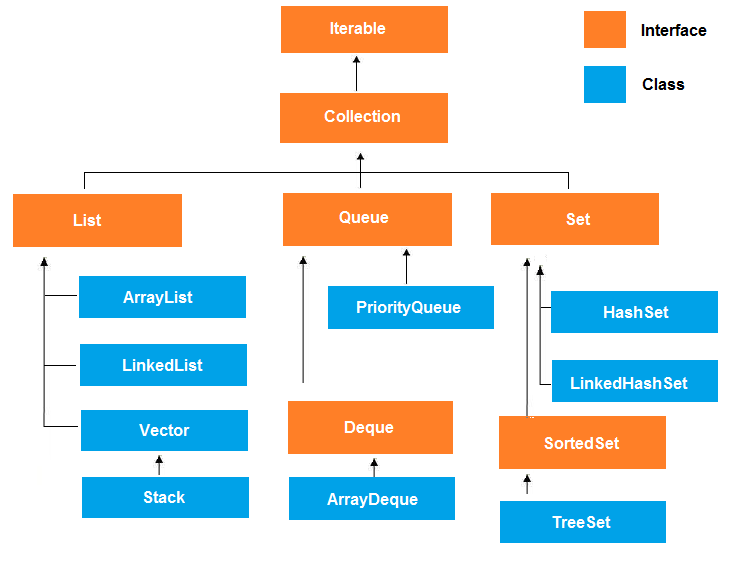
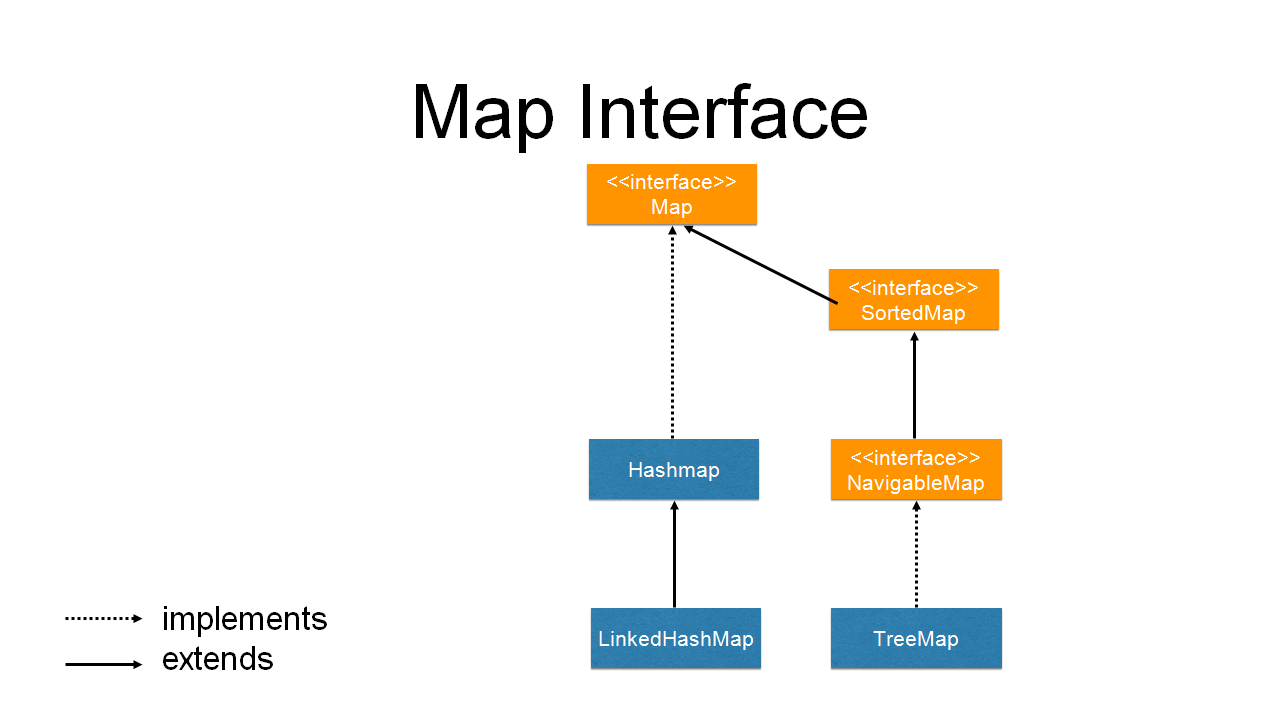
***ArrayList***

**Collections Framework hierarchy**





The Java Collections Framework is a collection of interfaces and classes which helps in storing and processing the data efficiently. This framework has several useful classes which have tons of useful functions which makes a programmer task super easy.

It is in memory data storage having multiple manipulation capabilities. There are two "groups" of interfaces: Collection's and Map's.

## 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.

***List (properties)***

* Elements are ordered using zero based index.
* Provides random access using index.
* Elements can be inserted & removed randomly using index.
* A list may contain duplicate element.
* A list may contain multiple null element.

***ArrayList (class)***

* This is resizable array. It can grow and shrink dynamically.
* Elements can be inserted and deleted randomly using index.
* If generics are not used then it can store any type of data.
* User can traverse an arraylist in both direction – forward and backward using ListIterator.
* ArrayList can hold multiple null elements.
* ArrayList can hold duplicate elements.
* Its time complexity performance is constant O(1) for get(), add() and for remove() O(n)
* It grows 50% of its size.

## How to create an ArrayList?

This statement creates an ArrayList with the name alist with type “String”. The type determines which type of elements the list will have.

List listA = new ArrayList();

List listB = new LinkedList();

List listC = new Vector();

List listD = new Stack();

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

By default you can put any Object into a List, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a List. Here is an example:

List<MyObject> list = new ArrayList<MyObject>();

This List can now only have MyObject instances inserted into it. You can then access and iterate its elements without casting them. Here is how it looks:

MyObject myObject = list.get(0);

for(MyObject anObject : list){

//do someting to anObject...

}

Since this list is of “String” type, the elements that are going to be added to this list will be of type “String”.

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

Similarly we can create ArrayList that accepts int elements.

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

## How to add elements to an ArrayList?

We add elements to an ArrayList by using add() method, this method has couple of variations, which we can use based on the requirement. For example: If we want to add the element at the end of the List then simply do it like this:

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

To add the element at the specified location in ArrayList, we can specify the index in the add method like this:

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

Lets write the complete code:

import java.util.\*;

class JavaExample{

public static void main(String args[]){

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

alist.add("Steve");

alist.add("Tim");

alist.add("Lucy");

alist.add("Pat");

alist.add("Angela");

alist.add("Tom");

//displaying elements

System.out.println(alist);

//Adding "Steve" at the fourth position

alist.add(3, "Steve");

//displaying elements

System.out.println(alist);

}

}

**Output:**

[Steve, Tim, Lucy, Pat, Angela, Tom]

[Steve, Tim, Lucy, Steve, Pat, Angela, Tom]

**Note:** Since the index starts with 0, index 3 would represent fourth position not 3.

## Insert All Elements From One List Into Another

It is possible to add all elements from one Java List into another List. You do so using the ListaddAll() method. The resulting List is the union of the two lists. Here is an example of adding all elements from one List into another:

List listSource = new ArrayList();

listSource.add("123");

listSource.add("456");

List listDest = new ArrayList();

listDest.addAll(listSource);

This example adds all elements from listSource into listDest.

The addAll() method takes a Collection as parameter, so you can pass either a List or [**Java Set**](http://tutorials.jenkov.com/java-collections/set.html) as parameter. In other words, you can add all elements from a List or Set into a List with addAll() .

## Get Elements From a Java List

You can get the elements from a Java List using the index of the elements. You can do so using either the get(int index) method. Here is an example of accessing the elements of a Java List using the element indexes:

List listA = new ArrayList();

listA.add("element 0");

listA.add("element 1");

listA.add("element 2");

//access via index

String element0 = (String) listA.get(0);

String element1 = (String) listA.get(1);

String element3 = (String) listA.get(2);

It is also possible to iterate the elements of a Java List in the order they are stored in internally. I will show you how to do that later in this Java List tutorial.

## Find Elements in a List

You can find elements in a Java List using one of these two methods:

* indexOf()
* lastIndexOf()

The indexOf() method finds the index of the first occurrence in the List of the given element. Here is an example finding the index of two elements in a Java List:

List list = new ArrayList();

String element1 = "element 1";

String element2 = "element 2";

list.add(element1);

list.add(element2);

int index1 = list.indexOf(element1);

int index2 = list.indexOf(element2);

System.out.println("index1 = " + index1);

System.out.println("index2 = " + index2);

Running this code will result in this output:

index1 = 0

index2 = 1

### Find Last Occurrence of Element in a List

The lastIndexOf() method finds the index of the last occurrence in the List of a given element. Here is an example showing how to find the index of the last occurrence of a given element in a Java List:

List list = new ArrayList();

String element1 = "element 1";

String element2 = "element 2";

list.add(element1);

list.add(element2);

list.add(element1);

int lastIndex = list.lastIndexOf(element1);

System.out.println("lastIndex = " + lastIndex);

The output printed from running the above Java example will be:

lastIndex = 2

The element element 1 occurs 2 times in the List. The index of the last occurrence is 2.

## Checking if List Contains Element

You can check if a Java List contains a given element using the List contains() method. Here is an example of checking if a Java List contains an element using the contains() method:

List list = new ArrayList();

String element1 = "element 1";

list.add(element1);

boolean containsElement =

list.contains("element 1");

System.out.println(containsElement);

The output from running this Java List example will be:

true

... because the List does actually contain the element.

To determine if the List contains the element, the List will internally iterate its elements and compare each element to the object passed as parameter. The comparison uses the [**Java equals**](http://tutorials.jenkov.com/java-collections/hashcode-equals.html) method of the element to check if the element is equal to the parameter.

Since it is possible to add null values to a List, it is actually possible to check if the List contains a null value. Here is how you check if a List contains a null value:

list.add(null);

containsElement = list.contains(null);

System.out.println(containsElement);

Obviously, if the input parameter to contains() is null, the contains() method will not use the equals() method to compare against each element, but rather use the == operator.

## How to remove elements from ArrayList?

We use remove() method to remove elements from an ArrayList, Same as add() method, this method also has few variations.

For example:

import java.util.\*;

class JavaExample{

public static void main(String args[]){

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

alist.add("Steve");

alist.add("Tim");

alist.add("Lucy");

alist.add("Pat");

alist.add("Angela");

alist.add("Tom");

//displaying elements

System.out.println(alist);

//Removing "Steve" and "Angela"

alist.remove("Steve");

alist.remove("Angela");

//displaying elements

System.out.println(alist);

//Removing 3rd element

alist.remove(2);

//displaying elements

System.out.println(alist);

}

}

**Output:**

[Steve, Tim, Lucy, Pat, Angela, Tom]

[Tim, Lucy, Pat, Tom]

[Tim, Lucy, Tom]

## Iterating ArrayList

In the above examples, we have displayed the ArrayList elements just by referring the ArrayList instance, which is definitely not the right way to displays the elements. The correct way of displaying the elements is by using an advanced for loop like this.

import java.util.\*;

class JavaExample{

public static void main(String args[]){

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

alist.add("Gregor Clegane");

alist.add("Khal Drogo");

alist.add("Cersei Lannister");

alist.add("Sandor Clegane");

alist.add("Tyrion Lannister");

//iterating ArrayList

for(String str:alist)

System.out.println(str);

}

}

**Output:**

Gregor Clegane

Khal Drogo

Cersei Lannister

Sandor Clegane

Tyrion Lannister

## ArrayList Example in Java

This example demonstrates how to create, initialize, add and remove elements from ArrayList. In this example we have an ArrayList of type “String”. We have added 5 String element in the ArrayList using the method add(String E), this method adds the element at the end of the ArrayList.

We are then adding two more elements in the ArrayList using method add(int index, String E), this method adds the specified element at the specified index, index 0 indicates first position and 1 indicates second position.

We are then removing the elements “Chaitanya” and “Harry” from the ArrayList and then we are removing the second element of the ArrayList using method remove(int index). Since we have specified the index as 1 (remove(1)), it would remove the second element.

import java.util.\*;

public class JavaExample {

public static void main(String args[]) {

/\* Creating ArrayList of type "String" which means

\* we can only add "String" elements

\*/

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

/\*This is how we add elements to an ArrayList\*/

obj.add("Ajeet");

obj.add("Harry");

obj.add("Chaitanya");

obj.add("Steve");

obj.add("Anuj");

// Displaying elements

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

for(String str:obj)

System.out.println(str);

/\* Add element at the given index

\* obj.add(0, "Rahul") - Adding element "Rahul" at first position

\* obj.add(1, "Justin") - Adding element "Justin" at second position

\*/

obj.add(0, "Rahul");

obj.add(1, "Justin");

// Displaying elements

System.out.println("ArrayList after add operation:");

for(String str:obj)

System.out.println(str);

//Remove elements from ArrayList like this

obj.remove("Chaitanya"); //Removes "Chaitanya" from ArrayList

obj.remove("Harry"); //Removes "Harry" from ArrayList

// Displaying elements

System.out.println("ArrayList after remove operation:");

for(String str:obj)

System.out.println(str);

//Remove element from the specified index

obj.remove(1); //Removes Second element from the List

// Displaying elements

System.out.println("Final ArrayList:");

for(String str:obj)

System.out.println(str);

}

}

**Output:**

Original ArrayList:

Ajeet

Harry

Chaitanya

Steve

Anuj

ArrayList after add operation:

Rahul

Justin

Ajeet

Harry

Chaitanya

Steve

Anuj

ArrayList after remove operation:

Rahul

Justin

Ajeet

Steve

Anuj

Final ArrayList:

Rahul

Ajeet

Steve

Anuj

## Methods of ArrayList class

In the above example we have used methods such as add() and remove(). However there are number of methods available which can be used directly using object of ArrayList class. Let’s discuss few important 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");

It will add the string bye to the 2nd index (3rd position as the array list starts with index 0) of array list.

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

obj.remove("Chaitanya");

This statement will remove the string “Chaitanya” from the ArrayList.

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

obj.remove(3);

It would remove the element of index 3 (4th element of the list – List starts with o).

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");

This would give the index (position) of the string Tom in the list.

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();

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

1. **import** java.util.\*;
2. **class** TestJavaCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

Output:

Ravi

Vijay

Ravi

Ajay

### **Iterating Collection through Iterator interface**

Let's see an example to traverse ArrayList elements using the Iterator interface.

1. **import** java.util.\*;
2. **class** ArrayList2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection1)

Ravi

Vijay

Ravi

Ajay

### **Iterating Collection through the for-each loop**

Let's see an example to traverse the ArrayList elements using the for-each loop

1. **import** java.util.\*;
2. **class** ArrayList3{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. //Traversing list through for-each loop
10. **for**(String obj:al)
11. System.out.println(obj);
12. }
13. }

Ravi

Vijay

Ravi

Ajay

### **Iterating Collection through remaining ways**

Let's see an example to traverse the ArrayList elements through other ways

1. **import** java.util.\*;
2. **class** ArrayList4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
10. System.out.println("Traversing list through List Iterator:");
11. //Here, element iterates in reverse order
12. ListIterator<String> list1=list.listIterator(list.size());
13. **while**(list1.hasPrevious())
14. {
15. String str=list1.previous();
16. System.out.println(str);
17. }
18. System.out.println("Traversing list through for loop:");
19. **for**(**int** i=0;i<list.size();i++)
20. {
21. System.out.println(list.get(i));
22. }
24. System.out.println("Traversing list through forEach() method:");
25. //The forEach() method is a new feature, introduced in Java 8.
26. list.forEach(a->{ //Here, we are using lambda expression
27. System.out.println(a);
28. });
30. System.out.println("Traversing list through forEachRemaining() method:");
31. Iterator<String> itr=list.iterator();
32. itr.forEachRemaining(a-> //Here, we are using lambda expression
33. {
34. System.out.println(a);
35. });
36. }
37. }

Traversing list through List Iterator:

Ajay

Ravi

Vijay

Ravi

Traversing list through for loop:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEach() method:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEachRemaining() method:

Ravi

Vijay

Ravi

Ajay

### **User-defined class objects in Java ArrayList**

Let's see an example where we are storing Student class object in an array list.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }
11. **import** java.util.\*;
12. **class** ArrayList5{
13. **public** **static** **void** main(String args[]){
14. //Creating user-defined class objects
15. Student s1=**new** Student(101,"Sonoo",23);
16. Student s2=**new** Student(102,"Ravi",21);
17. Student s2=**new** Student(103,"Hanumat",25);
18. //creating arraylist
19. ArrayList<Student> al=**new** ArrayList<Student>();
20. al.add(s1);//adding Student class object
21. al.add(s2);
22. al.add(s3);
23. //Getting Iterator
24. Iterator itr=al.iterator();
25. //traversing elements of ArrayList object
26. **while**(itr.hasNext()){
27. Student st=(Student)itr.next();
28. System.out.println(st.rollno+" "+st.name+" "+st.age);
29. }
30. }
31. }

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

# Java Iterator

Iterator is used for iterating (looping) various collection classes such as [HashMap](https://beginnersbook.com/2013/12/hashmap-in-java-with-example/" \o "HashMap in Java with Example" \t "_blank), [ArrayList](https://beginnersbook.com/2013/12/java-arraylist/" \o "ArrayList in java with example programs – Collections Framework" \t "_blank), [LinkedList](https://beginnersbook.com/2013/12/linkedlist-in-java-with-example/" \o "LinkedList in Java with Example" \t "_blank) etc. In this tutorial, we will learn what is iterator, how to use it and what are the issues that can come up while using it. Iterator took place of Enumeration, which was used to iterate legacy classes such as Vector. We will also see the differences between Iterator and Enumeration in this tutorial.

## Iterator without Generics Example

Generics got introduced in Java 5. Before that there were no concept of Generics.

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorDemo1 {

public static void main(String args[]){

ArrayList names = new ArrayList();

names.add("Chaitanya");

names.add("Steve");

names.add("Jack");

Iterator it = names.iterator();

while(it.hasNext()) {

String obj = (String)it.next();

System.out.println(obj);

}

}

}

**Output**:

Chaitanya

Steve

Jack

In the above example we have iterated ArrayList without using Generics. Program ran fine without any issues, however there may be a possibility of ClassCastException if you don’t use Generics (we will see this in next section).

## Iterator with Generics Example

In the above section we discussed about ClassCastException. Lets see what is it and why it occurs when we don’t use Generics.

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorDemo2 {

public static void main(String args[]){

ArrayList names = new ArrayList();

names.add("Chaitanya");

names.add("Steve");

names.add("Jack");

//Adding Integer value to String ArrayList

names.add(new Integer(10));

Iterator it = names.iterator();

while(it.hasNext()) {

String obj = (String)it.next();

System.out.println(obj);

}

}

}

Output:

ChaitanyaException in thread "main"

Steve

Jack

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

at beginnersbook.com.Details.main(Details.java:18)

In the above program we tried to add Integer value to the ArrayList of String but we didn’t get any compile time error because we didn’t use Generics. However since we type casted the integer value to String in the while loop, we got ClassCastException.

**Use Generics:**  
Here we are using Generics so we didn’t type caste the output. If you try to add a integer value to ArrayList in the below program, you would get compile time error. This way we can avoid ClassCastException.

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorDemo3 {

public static void main(String args[]){

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

names.add("Chaitanya");

names.add("Steve");

names.add("Jack");

Iterator<String> it = names.iterator();

while(it.hasNext()) {

String obj = it.next();

System.out.println(obj);

}

}

}

Note: We did not type cast iterator returned value[it.next()] as it is not required when using Generics.

## Difference between Iterator and Enumeration

An [iterator](https://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html)over a collection. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:  
1) Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.  
2) Method names have been improved. hasNext() method of iterator replaced hasMoreElements() method of enumeration, similarly next() replaced nextElement().

## ConcurrentModificationException while using Iterator

import java.util.ArrayList;

public class ExceptionDemo {

public static void main(String args[]){

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

books.add("C");

books.add("Java");

books.add("Cobol");

for(String obj : books) {

System.out.println(obj);

//We are adding element while iterating list

books.add("C++");

}

}

}

Output:

C

Exception in thread "main" java.util.ConcurrentModificationException

at java.util.ArrayList$Itr.checkForComodification(Unknown Source)

at java.util.ArrayList$Itr.next(Unknown Source)

at beginnersbook.com.Details.main(Details.java:12)

We cannot add or remove elements to the collection while using iterator over it.

**LinkedList**

***LinkedList (class)***

* This class implemented doubly linked list which can be used both List as well as Queue.
* User can insert data from both ends because each nodes have previous and next pointers.
* User can retrieve or delete elements either from head, tail or from middle of the nodes.
* Insertion and removal is faster than ArrayList, because no shifting is required.
* Retrieval of elements is very slow compare to ArrayList.
* LinkedList can be used as ArrayList, Queue, singly or doubly linked list.
* This can have null elements.
* This can have duplicate elements.
* Does not support random access.
* Its time complexity performance is constant O(n) for get(), add() O(1) and for remove() O(n)

LinkedList is a doubly-linked list implementation of the List and Deque interfaces. LinkedList allows for constant-time insertions or removals using iterators, but only sequential access of elements. In other words, LinkedList can be searched forward and backward.

## Example of LinkedList in Java

import java.util.\*;

public class LinkedListExample {

public static void main(String args[]) {

/\* Linked List Declaration \*/

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

/\*add(String Element) is used for adding

\* the elements to the linked list\*/

linkedlist.add("Item1");

linkedlist.add("Item5");

linkedlist.add("Item3");

linkedlist.add("Item6");

linkedlist.add("Item2");

/\*Display Linked List Content\*/

System.out.println("Linked List Content: " +linkedlist);

/\*Add First and Last Element\*/

linkedlist.addFirst("First Item");

linkedlist.addLast("Last Item");

System.out.println("LinkedList Content after addition: " +linkedlist);

/\*This is how to get and set Values\*/

Object firstvar = linkedlist.get(0);

System.out.println("First element: " +firstvar);

linkedlist.set(0, "Changed first item");

Object firstvar2 = linkedlist.get(0);

System.out.println("First element after update by set method: " +firstvar2);

/\*Remove first and last element\*/

linkedlist.removeFirst();

linkedlist.removeLast();

System.out.println("LinkedList after deletion of first and last element: " +linkedlist);

/\* Add to a Position and remove from a position\*/

linkedlist.add(0, "Newly added item");

linkedlist.remove(2);

System.out.println("Final Content: " +linkedlist);

}

}

Output:

Linked List Content: [Item1, Item5, Item3, Item6, Item2]

LinkedList Content after addition: [First Item, Item1, Item5, Item3, Item6, Item2, Last Item]

First element: First Item

First element after update by set method: Changed first item

LinkedList after deletion of first and last element: [Item1, Item5, Item3, Item6, Item2]

Final Content: [Newly added item, Item1, Item3, Item6, Item2]

**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();

**How to loop LinkedList in Java**

In the last tutorial we discussed LinkedList and it’s methods with example. Here we will see how to loop/iterate a LinkedList. There are four ways in which a LinkedList can be iterated –

1. For loop
2. Advanced For loop
3. Iterator
4. While Loop

#### Example:

In this example we have a LinkedList of String Type and we are looping through it using all the four mentioned methods.

package beginnersbook.com;

import java.util.\*;

public class LinkedListExample {

public static void main(String args[]) {

/\*LinkedList declaration\*/

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

linkedlist.add("Apple");

linkedlist.add("Orange");

linkedlist.add("Mango");

/\*for loop\*/

System.out.println("\*\*For loop\*\*");

for(int num=0; num<linkedlist.size(); num++)

{

System.out.println(linkedlist.get(num));

}

/\*Advanced for loop\*/

System.out.println("\*\*Advanced For loop\*\*");

for(String str: linkedlist)

{

System.out.println(str);

}

/\*Using Iterator\*/

System.out.println("\*\*Iterator\*\*");

Iterator i = linkedlist.iterator();

while (i.hasNext()) {

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

}

/\* Using While Loop\*/

System.out.println("\*\*While Loop\*\*");

int num = 0;

while (linkedlist.size() > num) {

System.out.println(linkedlist.get(num));

num++;

}

}

}

Output:

\*\*For loop\*\*

Apple

Orange

Mango

\*\*Advanced For loop\*\*

Apple

Orange

Mango

\*\*Iterator\*\*

Apple

Orange

Mango

\*\*While Loop\*\*

Apple

Orange

Mango

**Vector**

#### 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).

## Complete Example of Vector in Java:

import java.util.\*;

public class VectorExample {

public static void main(String args[]) {

/\* Vector of initial capacity(size) of 2 \*/

Vector<String> vec = new Vector<String>(2);

/\* Adding elements to a vector\*/

vec.addElement("Apple");

vec.addElement("Orange");

vec.addElement("Mango");

vec.addElement("Fig");

/\* check size and capacityIncrement\*/

System.out.println("Size is: "+vec.size());

System.out.println("Default capacity increment is: "+vec.capacity());

vec.addElement("fruit1");

vec.addElement("fruit2");

vec.addElement("fruit3");

/\*size and capacityIncrement after two insertions\*/

System.out.println("Size after addition: "+vec.size());

System.out.println("Capacity after increment is: "+vec.capacity());

/\*Display Vector elements\*/

Enumeration en = vec.elements();

System.out.println("\nElements are:");

while(en.hasMoreElements())

System.out.print(en.nextElement() + " ");

}

}

Output:

Size is: 4

Default capacity increment is: 4

Size after addition: 7

Capacity after increment is: 8

Elements are:

Apple Orange Mango Fig fruit1 fruit2 fruit3

## Commonly used methods of Vector Class:

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.