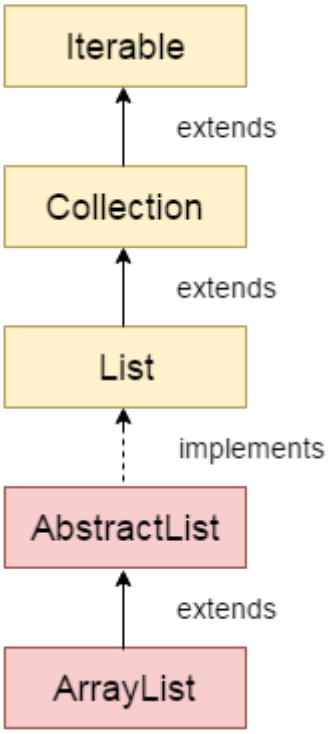
ArrayList Class



IQ. How an object like ArrayList holds data?
Collection in Java is simply a data structure. One Type follow binary, then one follow Tree or Linked List and so on.
Vector and ArrayList both follows the same data structure to hold unlimited data. Vector and ArrayList is like brother with little differences. Vector came in 1.2 and ArrayList copied all works.

- There is no capacity method in ArrayList().
- ArrayList default size is 10 and it grows 50% of initial capacity when needed "**Initial capacity/2*2**"
- Vector class **double** the Initial value when needed.

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

#. ArrayList, Vector classes are implemented using dynamically re-sizable arrays providing fast random access and list traversion very much like using an ordinary arrays.

#. ArrayList support dynamic arrays that can grow as needed that is ArrayList can dynamically increases or decreases in size.

#. ArrayList are created with an initial size when the size exceeded the collection is automatically enlarged.
If elements successfully added it returns **true** else **false**. This method is of Collection interface so all child **List** and **Set** will have this method.

#. Some important points about ArrayList class:

1. ArrayList is not synchronized.
2. ArrayList supports dynamic array which can grow as needed.
3. Size of ArrayList can be dynamically increased (via ensuseCapasity()) or decreased (trimSize()).
4. ArrayList are created with initial size of 10.
5. ArrayList can contain duplicate elements. ArrayList maintains insertion orders of the elements.
6. ArrayList is not synchronized collection hence it is not suitable to be used between multiple threads concurrently. If you want to be use ArrayList like data-structure in

multi-threaded environment, then you need to either use new **copyOnWriteArrayList**() or use **Collections**.**synchronizedList**() to create a synchronized list.

7. Former is part of concurrent collection package and much more suitable than the second one, but only useful when there are readers and only few writers. Since a new copy of ArrayList is created every time a write happens. It can be overkill if used in a write heavy environment. Second option is strictly synchronized collection, much like **Vector** or **HashTable,** but it is not scalable because once number of thread increased drastically contention became a huge issue.

8. **CopyOnWriteArrayList**() is recommended for concurrent multi-threading environment as it is optimized for multiple concurrent read and creates copy for the write operation. This was added in JDK 1.5. It's part of java.util.Concurrent package, along with **ConcurrentHashmap**.

9. When ArrayList gets full it creates another array and uses **System.arrayCopy**() to copy all elements from one array to another array. This is where insertion takes a lot of time.

10. Iterator and ListIterator of Java ArrayList are **fail-fast**. It means if ArrayList is structurally modified at any time after the Iterator is created in any way except through their Iterator's add() methods or remove() methods, the iterator will throw a **ConCurrentModificationException**. Thus in the case of concurrent modification, the Iterators fails quickly and clearly, that's why it is called fail-fast.

11. ConcurrentModificationException is not guaranteed and only thrown at best effort.

12. If we are creating synchronized list it's recommended to create while creating instance of underlying ArrayList to prevent accidental non-synchronized access to the list. It means don't first create non-synchronized ArrayList then convert it on synchronized. We must create it synchronized at first place.

13. An application can increase the capacity of an ArrayList instance before adding a large number of elements using the **ensureCapacity()** method operation. This may reduce amount of incremental reallocation due to incremental filing of ArrayList.
14. The size(), isEmpty, set(), iterator() and iteratorList() operation run in constant time because ArrayList is based on Array but adding or removing on element is costly as compared to LinkedList.

15. ArrayList class is enhanced in JDK1.5 to support Generics which added extra type-safety on ArrayList. It is recommended to use generics version of ArrayList to ensure that your ArrayList contains only specified type of element and avoid any ClassCastException.
16. If we set ArrayList reference (variable) to null in Java. All the elements inside ArrayList becomes eligible to garbage collection in Java. Provided there are no more strong reference exists for those object.

17. Always use **isEmpty ()** method to check if ArrayList is empty or not, instead of using **size () == 0** check. Former one is much more readable.

if(listOfElements.isEmpty()) { S.O.P(“Start Processing”); }

if(listOrders.size()) { S.O.P(“Start Processing”); }

Constructors of ArrayList:-

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Q. What if we make non-generic ArrayList.

Java collection framework was non-generic before **JDK 1.5**. Since 1.5, it is generic.
If we make non-generic ArrayList so we can put different-different type of object but while iterating we have to downcast from Object to Our class and here down-casting will be fail and it will throw **ClassCastException**.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time. In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

Vector Class:-

Big Sister of ArrayList, same mechanishm with little differences.

ArrayList and Vector both **implements List interface** and maintains insertion order.

Vector class has 3 constructor same as ArrayList and one extra **4th constructor:**

Vector(int capacity, int increament)

Vector class doubles the size when need to grow, that waste the memory so via 4th constructor we got provision to set the capacity as well as growth capacity.

Vector elements can access by Iterator as well as via Enumeration. Vector is legacy class, it means it is available Collection framework. Later it added to the Collection. Now Vector class hs it’s own legacy methods as well as ArrayList methods. Like ArrayList’s remove and it’s own old removeElement.

Difference between ArrayList and Vector Class?

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 another thread performing remove operation on ArrayList at the same time in a multi-threaded 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.

Resize: Both ArrayList and Vector can grow and shrink dynamically to maintain the optimal use of storage. However the way they resize is different. ArrayList grow by half of its size when resize. In the other hand Vector doubles the size of itself by default when grows.

Performance: ArrayList gives better performance as it is non-synchronized. Vector operation ogives 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.

Fail-Fast: If the collection (ArrayList, Vector etc.) gets structurally modified by any means, except add or remove methods of iterator, after creation of iterator then the iterator will throw **ConcurrentModificationException**. Structural modification refers to the addition or deletion of elements from the collection.

As per **Vector Javadoc** the Enumeration returned by Vector is not fail-fast. On the other side the iterator and ListIterator retuned by ArrayList are fail-fast.

Q. Who belongs to Collection framework?

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 use **copyOnWriteArrayList**() method which is a thread-safe variant of ArrayList.

Size Increment: In ArrayLIst we cannot define the increment size but Vector we can define the increment size.

Similarities: There are following similarities between those closes which are as follows.

- Both Vector and ArrayList use grow able array data structure.
- The Iterator and ListIterator returned by these classes (Vector and ArrayList) are fail-fast.
- They both are ordered collection classes as they maintain the elements insertion order.
- Vector & ArrayList both allows duplicate and null values. They both grow shrink automatically when overflow and deletion happens.

Q. 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 then Vector is your best 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 better choice as performance will be improved because of the concurrent processed.

1. Programs: Adding Elements

```
import java.util.ArrayList;      // required for ArrayList
import java.util.Iterator;

public class ArrayListDemo{

    public static void main(String...args){

        // Generic ArrayList, create object.
        ArrayList<String> names = new ArrayList<> ();
        names.add("Ahmad");       // add method adds the element in the ArrayList
        names.add("Sayeed");
        names.add("Asmal");
        // names.add('Y');

        // names.add(10); we have fixed type “String”. No other data type can
            store.

        // Let's print the size/length of the ArrayList
        System.out.println("Size/Length is: " + names.size());

        //Let's remove the element from ArrayList
        names.remove(1);  // it will remove "Sayeed"

        // Let's print the size/length after removal
        System.out.println("After Removal Length is: " + names.size());

        // Let's iterate the elements from ArrayList
        Iterator<String> it = names.iterator();
        while(it.hasNext()){
            String s = it.next();     // Auto UN-Boxing
            System.out.println(s);
        }


        /* Non-Generic ArrayList */
        System.out.println("\n=======Non Generic ArrayList=======\n");

        ArrayList age = new ArrayList();
        age.add(25);              // Auto Boxing

        // Providing wrapper class that will hold in Object
        age.add(new Integer(26));
        age.add(27);

        // As here no type defined. But it will raise problem while iterating
        age.add("MY Age");
        System.out.println("Size/Length is: " + age.size());
```

```
        age.remove(2);
        System.out.println("After Removal Length is: " + age.size());

        // Iterating: It is not generic so it won't downcast implicitly, we have
        to do this explicitly and this is the problem.

        Iterator itr = age.iterator();
        while(itr.hasNext()){
            Integer z = (Integer) itr.next(); // String cannot be cast to Integer
            //Integer z = itr.next();    // Object cannot be converted to Integer
            System.out.println(z);
        }

        /*When while loop reach to index 3 means at string value, it will throw
        ClassCastException */
        // Also when use Non-Generic ArrayList, compiler shows warning
    }
}
```

Program 2: Ways To Add and Get Elements From Collections

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorWays{

    public static void main(String...args){
        // 1.
        System.out.println("\n====Via iterator() method====");
        via_iterator_function();

        // 2.
        System.out.println("\n====Via toArray() method====");
        via_toArray_function();

        // 3.
        System.out.println("\n====Via Advance for loop (foreach)====");
        via_advance_for_loop();

        // 4.
        System.out.println("\n====Via get() function====");
        via_get_function();
    }


    public static void via_iterator_function(){
        ArrayList<Integer> al = new ArrayList<> ();

        int[] x = {1, -1, 2,  -2, 3, -3, 4, -4};
        // Add Via Loop
        for(int i=0; i<x.length; i++){
            al.add(x[i]);
        }
        System.out.println(al);

        // Get Elements via iterator()
```

```
    Iterator<Integer> itr = al.iterator();

    while(itr.hasNext()){
        Integer i = itr.next();   // Auto UN-boxing and down-casting.
        if(i < 0) itr.remove();
        // Integer i = (Integer ) itr.next();   //Down-casting
    }
    System.out.println("After Removal: " + al);
}


public static void via_toArray_function(){

    ArrayList<Integer> al = new ArrayList<> ();
    int[] x = {1, -1, 2,  -2, 3, -3, 4, -4};
    // Add Via Loop
    for(int i=0; i<x.length; i++){
        al.add(x[i]);
    }
    System.out.println(al);

    // Get Elements via toArray() method
    System.out.print("Using toArray(): ");
    Object[] o = al.toArray();

    for(int i=0; i < o.length; i++){
        Integer z = (Integer) o[i];      // Down-casting required
        System.out.print(z.intValue() + " ");
        // System.out.println(z+ " ");
    }
    System.out.println();
}


public static void via_advance_for_loop(){
    ArrayList<Integer> al = new ArrayList<> ();
    int[] x = {1, -1, 2,  -2, 3, -3, 4, -4};
    // Add Via Loop
    for(int i=0; i<x.length; i++){
        al.add(x[i]);
    }
    System.out.println(al);

    System.out.print("Using foreach: ");
    for(Integer it: al){
        System.out.print(it.intValue() + " ");
        // System.out.println(it+ " ");
    }
}

public static void via_get_function(){
    System.out.println();
    ArrayList<String> arrList = new ArrayList<>();
    arrList.add("String 1");
    arrList.add("String 2");
    arrList.add("String 3");
    arrList.add("String 4");
```

```
        // ArrayList To Array

        // Create an array of ArrayList Size
        String[] s1 = new String[arrList.size()];

        for(int i=0; i<arrList.size(); i++){
            s1[i] = arrList.get(i);
        }

        // Displaying Array s1 elements

        for(String s: s1){
            System.out.println(s);
        }

    }
}
```

Program 3: Q. What happens when create ArrayList of "ArrayList Type"

```
import java.util.ArrayList;

public class ListDemo3{

    public static void main(String... args){

        // create ArrayList of "ArrayList TYpe"
        ArrayList <ArrayList> al = new ArrayList<> ();
        System.out.println("Size OF al is: " + al.size() );

        // New ArrayList, Type String
        ArrayList<String> s1 = new ArrayList<> ();
        s1.add("Ahmad");
        s1.add("Asmal");
        s1.add("Sayeed");

        al.add(s1);
        System.out.println("After adding s1: " + al.size());

        // Just added own ArrayList reference id to the own ArrayList as an
    element.
        al.add(al);
        System.out.println("After adding itself al: " + al.size());

        System.out.println(al.get(1).get(0));       // will get s1 reference
    }
}
```

5. Overloaded add(Integer index, Object go)

```
// All Collection methods come to List. List class has also overloaded
many number of methods.

import java.util.ArrayList;

public class AddMethod{

    public static void main(String...a){

        // Create Generic ArrayList object, type String
        ArrayList<String> al = new ArrayList<>();
        al.add("HI");
        al.add("Hello");
        al.add("Welcome");
        al.add("Test STring");

        System.out.println(al);

        // Using the overloaded method of List add(Integer index, Object go)
        al.add(0, "New String");
        al.add(0, "Good Bye");

        System.out.println("More Elements: " + al);
    }
}
```

6. Program: Serialize and De-Serialize Predefined class "ArrayList"

```
import java.util.ArrayList;
import java.util.Iterator;
import java.io.*;

public class ArrayListSerialization{

    public static void main(String...args){

        ArrayList<String> al = new ArrayList<> ();
        al.add("Hello");
        al.add("Hi");
        al.add("Welcome");

        // Serialization
        try(FileOutputStream fos = new FileOutputStream("myFile");
    ObjectOutputStream oos = new ObjectOutputStream(fos); ){
            // write Object into stream
            oos.writeObject(al);
        }catch(Exception e){  e.printStackTrace();  }

        // DeSerialization
        System.out.println("After de-serialization");

        try(FileInputStream fin = new FileInputStream("myFile");
    ObjectInputStream oin = new ObjectInputStream(fin); ){
```

```
            // read the object from stream
            // Down cast into Array
            ArrayList alNew = (ArrayList) oin.readObject();
            Iterator it = alNew.iterator();
            while(it.hasNext()){
                System.out.println(it);
            }
            System.out.println(alNew);
        }catch(Exception e){  e.printStackTrace();  }
    }
}
```

// Serialize and De-Serialize With User-Defined Class

```
import java.util.ArrayList;
import java.util.Iterator;
import java.io.*;

public class EmpSerialization{

    public static void main(String...args){

        ArrayList<Emp> al = new ArrayList<> ();
        al.add(new Emp(10));
        al.add(new Emp(20));
        al.add(new Emp(30));

        // Serialization
        try(FileOutputStream fos = new FileOutputStream("myFile1");
    ObjectOutputStream oos = new ObjectOutputStream(fos); ){
            // write Object into stream
            oos.writeObject(al);
        }catch(Exception e){  e.printStackTrace();  }

        // DeSerialization
        System.out.println("After de-serialization");

        try(FileInputStream fin = new FileInputStream("myFile1");
    ObjectInputStream oin = new ObjectInputStream(fin); ){

            // read the object from stream
            // Down cast into Array
            Object alNew1 = oin.readObject();
            ArrayList alNew = (ArrayList) alNew1;
             Iterator<Emp> it = alNew.iterator();
            while(it.hasNext()){
                Emp e = it.next();
                System.out.println(e.x);
            }

        }catch(Exception e){  e.printStackTrace();  }
    }
}
```

```java
final class Emp implements Serializable{
    private final static long serialVersionUID = 456L;
    final int x;

    //Constructor
    Emp(int x){
        this.x = x;
    }
}
```

7 Program: // ArrayList To ArrayList

```java
import java.util.Iterator;
import java.util.ArrayList;

public class ArrayListToArray{

    public static void main(String...a){

        ArrayList<String> arrList = new ArrayList<>();
        arrList.add("String 1");
        arrList.add("String 2");
        arrList.add("String 3");
        arrList.add("String 4");

        // ArrayList To Array
        // Create an array of ArrayList Size
        String[] s1 = new String[ arrList.size() ];

        for(int i=0; i<arrList.size(); i++){
            s1[i] = arrList.get(i);
        }

        // Displaying Array s1 elements
        for(String s: s1){
            System.out.println(s);
        }
    }
}
```