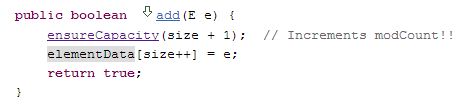
1. **Creates the empty list with initial capacity**  
     
     
    1.  List list = new ArrayList();

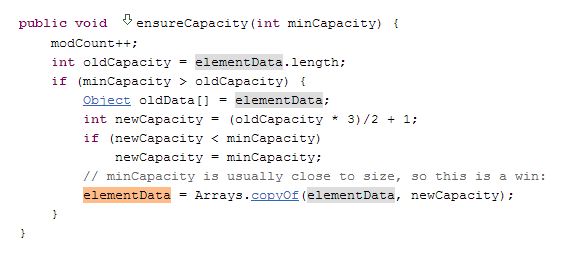
When we create ArrayList this way , the default constructor of the ArrayList class is invoked. It will create internally an array of Object with default size set to 10.  
  
 2.  List list = new ArrayList(20);  
  
When we create ArrayList this way , the  ArrayList will invoke the constructor with the integer argument. It will create internally an array of Object . The size of the Object[] will be equal to the argument passed in the constructor . Thus when above line of code is executed ,it  creates an Object[] of capacity 20.

**1>How does the size of Arraylist increases automatically? Could you share the code?**

**Ans=** when someone tries to add an object to the arraylist, Java checks to ensure that there is enough capacity in the existing array to hold the new object. If not, a new array of a greater size is created, the old array is copied to new array using Arrays.copyOf and the new array is assigned to the existing array. Look at the code below .

[](http://vitalflux.com/wp-content/uploads/2014/02/ArrayList-Add-Method.jpg)

*ArrayList Add Method*

[](http://vitalflux.com/wp-content/uploads/2014/02/ensureCapacity-Method-to-Handle-ArrayList-Size.jpg)

*ensureCapacity Method to Handle ArrayList Size*

Till Java 6  
  
 **int** newCapacity = (oldCapacity \* 3)/2 + 1;  
  
  
(Update) From Java 7  
  
     **int** newCapacity = oldCapacity + (oldCapacity >> 1);

Pay attention to the fact that a new array is created; Objects from old array is copied to the new array and the new array is assigned to the existing array member variable.

Q=vector internal implementation?

Ans=

@SuppressWarnings("unchecked")

**public** **synchronized** E **get**(**int** index) {

   //if index is negative or greater than size of size, we throw Exception.

**if** ( index <0 || index>= size) {

**throw** **new** IndexOutOfBoundsException("Index: " + index + ", Size " + index);

   }

**return** (E) elementData[index]; //return value on index.

 }

**public** **synchronized** Object **remove**(**int** index) {

**if** ( index <0 || index>= size) { //if index is negative or greater than size of size, we throw Exception.

**throw** **new** IndexOutOfBoundsException("Index: " + index + ", Size " + index);

   }

   Object removedElement=elementData[index];

**for**(**int** i=index;i<size - 1;i++){

      elementData[i]=elementData[i+1];

   }

   size--;   //reduce size of VectorCustom after removal of element.

**return** removedElement;

 }

**public** **synchronized void** **add**(E e) {

**if** (size == elementData.length) {

     ensureCapacity(); //increase current capacity of list, make it double.

   }

   elementData[size++] = e;

 }

 /\*\*

  \* method increases capacity of list by making it double.

  \*/

**private** **void** **ensureCapacity**() {

**int** newIncreasedCapacity = elementData.length \* 2;

   elementData = Arrays.*copyOf*(elementData, newIncreasedCapacity);

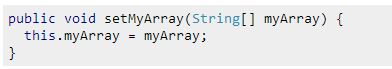
 }

Q= **When would you use ArrayList and when LinkedList?**

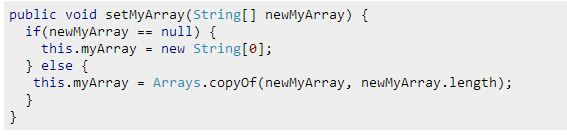
Ans=  one would want to use ArrayList in the cases where there is a greater need to access the element rather than insertion or deletion. On the other hand, one would want to use the LinkedList when there is a greater need for insertion and deletion and not much use of the accessing the element at a particular index. This is mainly because the **worst time complexity of accessing an element/object** in an ArrayList is always “**1″** whereas in LinkedList it can be**“n”**. **In case of addition or deletion of element** in an ArrayList, there is an operation such as System.arraycopy is involved. This is quite an expensive operation and thus, in usecases where frequent insertion and deletion of elements are involved, **LinkedList is preferred** where it is all about adding or removing the node and re-linking the existing node.

**Q=While passing an ArrayList to a method or returning an ArrayList from a method, when is it considered to be a security violation? How to fix this problem?**

When one passes the array to a method, if array is assigned to the member variable directly without making a copy of it, it could lead to a scenario that the original array when changed by the caller will also end up changing the array passed to the method. Look at the code below to see when it is security violation and subsequently how to fix the same.

[](http://vitalflux.com/wp-content/uploads/2014/02/Array-Stored-Directly-Considered-as-Security-Violation.jpg)

*Array Stored Directly – Considered as Security Violation*

[](http://vitalflux.com/wp-content/uploads/2014/02/Copy-the-Array-as-a-Fix-for-the-Security-Violation.jpg)

*Copy the Array as a Fix for the Security Violation*

Pay attention the fact that a copy of newMyArray is created and assigned to the member variable, myArray.

**Q=How do you copy one ArrayList to another? Could you share the code?**

Following are different techniques for copying an ArrayList object to another ArrayList:

1. Use clone() method of ArrayList
2. Use ArrayList constructor object such as following:

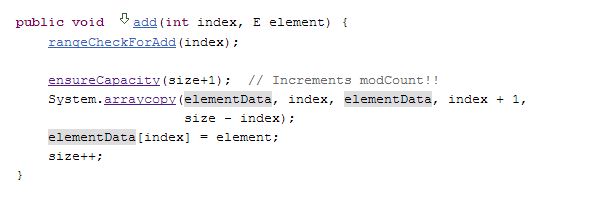
such as following: ArrayList myObject = new ArrayList<Object>(myTempObject);

1. use Collection.copy method

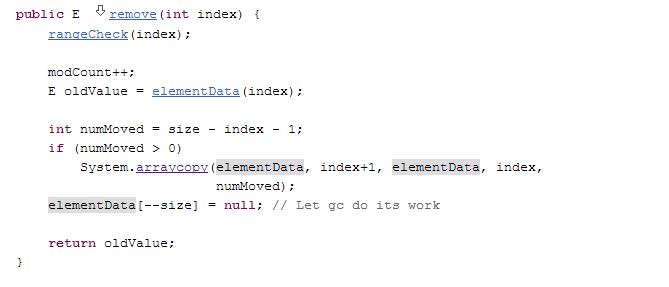
Note that option 1 and 2 creates a shallow copy.

**Q=How does the addition and deletion of an object at any index happens in ArrayList? Is it expensive? Explain?**

In case of addition/deletion of elements in an ArrayList, there is an operation such as System.arraycopy is involved. This is quite an expensive operation and for usecases like these where frequent insertion and deletion are required, one may want to choose other Java collection such as LinkedList. Take a look at the code below:

[](http://vitalflux.com/wp-content/uploads/2014/02/Adding-an-element-in-ArrayList-at-index-i.jpg)

*Adding an element in ArrayList at index i*

[](http://vitalflux.com/wp-content/uploads/2014/02/Removing-an-element-in-an-ArrayList-at-index-i.jpg)

*Removing an element in an ArrayList at index i*

### Q= difference between ArrayList and Vector

Ans=

But there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the  array size if total number of element exceeds  than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized  i.e. in multithreading environment,  it will hold the other threads in  runnable or non-runnable state  until current thread releases  the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration**  interface to traverse  the elements. But it can use Iterator also. |

### Q= difference between ArrayList and LinkedList?

Ans=

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to  store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than  ArrayList because it uses doubly linked list so  no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue**  both because it implements List and Deque  interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

**Q=Diffrence between Iterator and ListIterator?**

## Iterator vs ListIterator

1) Iterator is used for traversing List and Set both.

We can use ListIterator to traverse List only, we cannot traverse Set using ListIterator.

2) We can traverse in only forward direction using Iterator.

Using ListIterator, we can traverse a List in both the directions (forward and Backward).

3) We cannot obtain indexes while using Iterator

We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.

4) We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.

We can add element at any point of time while traversing a list using ListIterator.

5) We cannot replace the existing element value when using Iterator.

By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods.

6) Methods of Iterator:

* hasNext()
* next()
* remove()

Methods of ListIterator:

* add(E e)
* hasNext()
* hasPrevious()
* next()
* nextIndex()
* previous()
* previousIndex()
* remove()
* set(E e)

# Q= How to empty an ArrayList in Java?

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

As you can both the methods did the same job, they emptied the ArrayList. It’s time to determine which method gives good performance.

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)

Q=How to get subList of List?

Ans= List subList(int fromIndex, int toIndex)

# Q= Remove all elements from Vector?

Ans= vector\_obj.clear();

### Q= difference between Iterator and Enumeration

Only major difference between Enumeration and iterator is Iterator has a remove() method while Enumeration doesn't. Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects, where as by using Iterator we can manipulate the objects like adding and removing the objects from collection e.g. Arraylist.

Also **Iterator**is more secure and safe as compared to **Enumeration**because it  does not allow other thread to modify the collection object while some thread is iterating over it and throws **ConcurrentModificationException.**This is by far most important fact for me for deciding between Iterator vs Enumeration in Java.

### Q= difference between List and Set

* 1. List in Java allows duplicates while **Set doesn't allow any duplicate**. If you insert duplicate in Set it will replace the older value. Any implementation of Set in Java will only contains unique elements.
  2. Another significant difference between List and Set in Java is order. **List is an Ordered Collection** while Set is an unordered Collection. List maintains **insertion order of elements**
  3. Set uses [equals() method](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) to check uniqueness of elements stored in Set, while SortedSet uses [compareTo() method](http://javarevisited.blogspot.com/2011/11/how-to-override-compareto-method-in.html) to implement natural sorting order of elements.
  4. List allows any number of null values. Set can have only a single null value at most.
  5. [ListIterator](https://beginnersbook.com/2014/06/listiterator-in-java-with-examples/) can be used to traverse a List in both the directions(forward and backward) However it can not be used to traverse a Set. We can use [Iterator](https://beginnersbook.com/2014/06/java-iterator-with-examples/) (It works with List too) to traverse a Set.
  6. List interface has one legacy class called [Vector](https://beginnersbook.com/2013/12/vector-in-java/)whereas Set interface does not have any legacy class.
  7. List implementations: [ArrayList](https://beginnersbook.com/2013/12/java-arraylist/), [LinkedList](https://beginnersbook.com/2013/12/linkedlist-in-java-with-example/) etc.

Set implementations: [HashSet](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/), [LinkedHashSet](https://beginnersbook.com/2013/12/linkedhashset-class-in-java-with-example/), [TreeSet](https://beginnersbook.com/2013/12/treeset-class-in-java-with-example/) etc.

### Q= Difference between TreeSet, LinkedHashSet and HashSet in Java? Ans= 1. Implementation

All these 3 implement the **Set**interface.  
**HashSet**is backed by a hash table (actually a HashMap instance).  
**LinkedHashSet**uses Hash table and linked list. It maintains a doubly-linked list running through all of its entries, thus maintaining the order of insertion.  
**TreeSet**is a NavigableSet implementation based on a TreeMap.

### 2. Ordering

**HashSet**does not guarantee any order in which the elements are stored.  
**LinkedHashSet**maintains the order in which the elements are inserted.  
**TreeSet**sorts the elements in its natural ordering and ascending. You can provide a comparator to sort the elements stored in the TreeMap

### 3. Null values as Entries

**HashSet**and **LinkedHashSet**both allow **Null**elements to be added.  
**Null**is not allowed in a TreeSet unless you specify a comparator that can handle a **Null**.

### 4. Performance

**HashSet**are the fastest in terms of basic opertaions like add, remove or search elements.  
**LinkedHashSet**can be a little slower as they maintain a linked list which causes some performance overhead. But we never saw a major performance difference between HashSet and LinkedHashSet.  
**TreeSet**are the slowest in comparision with the other two, due to the sorting done.

# Q= **Difference between HashMap and Hashtable**

Ans=

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe  and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or**  **value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and  can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator**  **and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

### Q= between Comparable and Comparator

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**.  In other words, we can sort the collection on  the basis of multiple elements such as id, name  and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class**  i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort  elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |

### What is the advantage of Properties file?

If you change the value in properties file, you don't need to recompile the java class. So, it makes the application easy to manage.

### 14) What does the hashCode() method?

The hashCode() method returns a hash code value (an integer number).

The hashCode() method returns the same integer number, if two keys (by calling equals() method) are same.

But, it is possible that two hash code numbers can have different or same keys.

### 19) What is the Dictionary class?

The Dictionary class provides the capability to store key-value pairs.

### 20) What is the default size of load factor in hashing based collection?

The default size of load factor is **0.75**. The default capacity is computed as initial capacity \* load factor. For example, 16 \* 0.75 = 12. So, 12 is the default capacity of Map.

# **How to make ArrayList Read Only?**

Ans= List<String>unmodifiableList= Collections.unmodifiableList(fruitList);

# **How to reverse ArrayList in Java?**

Ans= Collections.reverse(l);

3. What is the difference between fail-fast and fail-safe Iterators?  
  
Ans= Fail-fast Iterators throws ConcurrentModificationException when one [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) is iterating over collection object and other thread structurally modify Collection either by adding, removing or modifying objects on underlying collection. They are called fail-fast because they try to immediately throw Exception when they encounter failure. On the other hand[fail-safe Iterators](http://javarevisited.blogspot.com/2011/10/java-iterator-tutorial-example-list.html) works on copy of collection instead of original collection  
  
1) Fail-fast Iterator throws ConcurrentModfiicationException as soon as they detect any structural change in collection during iteration, basically which changes the modCount variable hold by Iterator. While fail-fast iterator doesn't throw CME.

2) Fail-fast iterator traverse over original collection class while fail-safe iterator traverse over a copy or view of original collection. That's why they don't detect any change on original collection classes and this also means that you could operate with stale value.  
  
3) Iterators from Java 1.4 Collection classes e.g. ArrayList, HashSet and Vector are fail-fast while Iterators returned by concurrent collection classes e.g. [CopyOnWriteArrayList](http://java67.blogspot.com/2012/09/what-is-copyonwritearraylist-in-java-example-vs-arraylist.html) or [CopyOnWriteArraySet](http://javarevisited.blogspot.sg/2014/06/how-to-use-copyonwritearrayset-in-java-example-tutorial.html) are fail-safe.  
  
4) Iterator returned by synchronized Collection are fail-fast while iterator returned by concurrent collections are fail-safe in Java.  
  
5) Fail fast iterator works in live data but become invalid when data is modified while fail-safe iterator are weekly consistent.

### The difference between ConcurrentHashMap and Collections.synchronizedMap

# Ans= ConcurrentHashMap does not allow null keys or null values while synchronized HashMap allows one null key. Q=[**How linkedhashmap maintains insertion order**](https://stackoverflow.com/questions/20171999/how-linkedhashmap-maintains-insertion-order)?

Ans=<http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html>.

Idea behind implementation is quite simple. It extends regular hashMap (so it has all hashMap goodies) but also builds double linked list when adding elements.

(entries are also extended from the HashMap.Entry so they have pointers to after and before) So all entries are ordered HEAD -> Entry1 <-> Entry2 ... <-- TAIL

and at the same time kept in standard HashSet (i assume you are familiar with implementation).

Now when iterating It Linked list of entries is used.

# Q=**Working of Collections.synchronizedMap**

* Ans=Collections class defines a private nested static class named SynchronizedMap.
* SynchronizedMap has two instance variables **mutex** and **m.**
* **mutex**serves the purpose of a mutex variable in working of [Collections.synchronizedMap](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedMap-java.util.Map-) and **m**server the purpose of holding original collection.

This is how SynchronizedMap is defined roughly.

private static class SynchronizedMap<K,V>

implements Map<K,V>, Serializable {

private final Map<K,V> m; // references original map

final Object mutex; // server the purpose of mutex in sychronized methods.

}

package com.thejavageek.collections;

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class Demo {

public static void main(String[] args) {

Map<Integer, Object> map = new HashMap<Integer, Object>();

Map synchronizedMap = Collections.synchronizedMap(map);

synchronizedMap.put(1, null);

}

}

* Note that we are passing a **map** created as **HashMap**to **Collections.synchronizedMap(map);**
* It instantiates **SynchronizedMap** and passes **m**which refers to object referred by **map.**So, **m**is backing **map** now.
* **mutex**refers to current **SynrozniedMap** instance internally. We can see the same in source code itself.

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {

        return new SynchronizedMap<>(m);

    }

## **SynchronizedMap Constructor:**



|  |  |
| --- | --- |
| 1  2  3  4 | SynchronizedMap(Map<K,V> m) {      this.m = Objects.requireNonNull(m);      mutex = this;  } |

## **Calling map.put() method:**

Note that **Collections.synchronizedMap** returns an instance of type **SynchronizedMap.**so when you call **map.put(1,null);** due to polymorphism, SynchronizedMap.put() method is called which is synchronized on **mutex,**i.e. current instance of **SynchronizedMap.**



|  |  |
| --- | --- |
| 1  2  3 | public V put(K key, V value) {       synchronized (mutex) {return m.put(key, value);}  } |

Note that value is being put into original **map**backed by **m**which is synchronized on **mutex**and in code, it is being referred by **synchronizedMap** reference variable as shown below.

Java



|  |  |
| --- | --- |
| 1 | Map synchronizedMap = Collections.synchronizedMap(map); |

Q= [**Why do I need to override the equals and hashCode methods in Java?**](https://stackoverflow.com/questions/2265503/why-do-i-need-to-override-the-equals-and-hashcode-methods-in-java)

**Ans=**

<https://www.geeksforgeeks.org/override-equalsobject-hashcode-method/>

<https://www.geeksforgeeks.org/iterator-vs-foreach-in-java/>