**java.util.concurrent.CopyOnWriteArrayList<E>**

**public class CopyOnWriteArrayList<E> extends Object implements List<E>, RandomAccess, Cloneable, Serializable**

* A thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.
* This is ordinarily too costly, but may be more efficient than alternatives when traversal operations vastly outnumber mutations, and is useful when you cannot or don't want to synchronize traversals, yet need to preclude interference among concurrent threads.
* The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException.
* The iterator will not reflect additions, removals, or changes to the list since the iterator was created. Element-changing operations on iterators themselves (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.
* All elements are permitted, including null.
* Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a CopyOnWriteArrayList happen-before actions subsequent to the access or removal of that element from the CopyOnWriteArrayList in another thread.

Here are few points about CopyOnWriteArrayList:

* As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at certain point both will synchronized automatically which is takes care by JVM. Therefore there is no effect for threads which are performing read operation.
* It is costly to use because for every update operation a cloned copy will be created. Hence CopyOnWriteArrayList is the best choice if our frequent operation is read operation.
* It extends object and implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E> and RandomAccess
* The underlined data structure is grow-able array.
* It is thread-safe version of ArrayList.
* Insertion is preserved, duplicates are allowed and heterogeneous Objects are allowed.
* The main important point about CopyOnWriteArrayList is Iterator of CopyOnWriteArrayList can not perform remove operation otherwise we get Run-time exception saying UnsupportedOperationException.

**Difference between ArrayList and CopyOnWriteArrayList in Java**

**Answer: -**

* **Synchronization:** ArrayList is not synchronized. CopyOnWriteArrayList is synchronized. Synchronization means at a time only one thread can access the object.
* **Performance:**  ArrayList is faster as it is not synchronized. That means many threads can execute the same piece of code simultaneously. In comparison, CopyOnWriteArrayList is slower.
* **Fail-fast vs Fail-safe:** Iterators returned by ArrayList's iterator and listiterator methods are fail-fast. CopyOnWriteArrayList uses fail-safe iterator. We have already shared the fail-fast vs fail-safe iterators in java with example
* **ConcurrentModificationException :** ArrayList can throw ConcurrentModificationException while CopyOnWriteArrayList cannot .
* **Added in java version:** ArrayList class was added in java version 1.2, while CopyOnWriteArrayList class was added in java version 1.5 (or java 5) .
* **Package:**  ArrayList class is present in java.util package , while CopyOnWriteArrayList class is present in java.util.concurrent package.

**Similarities between ArrayList and CopyOnWriteArrayList**

* **Permit null values:** Both ArrayList and CopyOnWriteArrayList permits null.
* **Java Collections Framework:** Both classes are part of the Java Collections Framework.
* **Clone method:** Both classes clone method return the shallow copy of the original object.

**When to prefer CopyOnWriteArrayList over ArrayList**

**Answer: -**CopyOnWriteArrayList is a thread-safe variant of ArrayList in which all mutative operations (add, set and so on) are implemented by making a fresh copy of the underlying array. So, if your code involves multi-threading tasks then prefer CopyOnWriteArrayList over ArrayList . Otherwise, always use ArrayList .

# **SynchronizedList v/s CopyOnWriteArrayList:**

|  |  |
| --- | --- |
| **CopyOnWriteArrayList** | **SynchronizedList** |
| **CopyOnWriteArrayList** is newly introduced **thread-safe** class (i.e.; synchronized) | This is **thread-safe** version of List i.e.; any List implemented classes like ArrayList or LinkedList can be **converted** into synchronized List (thread-safe) |
| **Multiple threads** are allowed to operate on **CopyOnWriteArrayList**, as it works on separate **cloned copy** for **update/modify** operations | Only **one thread** is allowed to operate on synchronized list, by **locking over complete list** object |
| While one thread **iterating** **CopyOnWriteArrayList** object, other threads happily can **modify**, as it works on separate cloned copy  And it **never** throws **ConcurrentModificationException** | While one thread **iterating** List object, if any other threads tries to **modify** the same List object then **ConcurrentModificationException** is thrown |
| That’s it is **fail-safe iterator** | That’s it is **fail-fast iterator** |
| There is **no such restriction** while iterating CopyOnWriteArrayList;  We can safely iterate **outside synchronized block** | While **iterating** synchronized List, make sure to iterate inside **synchronized block**;  Otherwise we may face **non-deterministic behavior** |
| **Iterator of CopyOnWriteArrayList** can perform **read operation** safely; while iterating through **COWAL** items  But as soon as, **remove** operation is performed, compiler throws **UnsupportedOperationException** | **Iterator** of **List** can perform both **read** and **remove** operations; while iterating through **List** elements |
| This is introduced in **Java 1.5** version | This is introduced in original collection framework in **Java 1.2** version |

### **When to use SynchronizedList ?**

* This is generally used to **convert** list object into **thread-safe** list object
* But only one thread is allowed to operate on list object, as **lock** is acquired **over complete list** object
* So, **performance degrades** comparatively in a multi-threaded environment
* So, **use this only** when it is required to convert into thread-safe version of List object
* Otherwise, mature and latest **CopyOnWriteArrayList** can be used efficiently

### **When to use CopyOnWriteArrayList ?**

* This is the **best suit to store elements as per insertion order** in a multi-threaded environment
* Where there are **more number of read** operation and **very less update**/modify operation
* Because for every update/modify operations, a **new separate cloned copy** is created
* And there is **overhead on JVM** to allocate **memory** & **merging** cloned copy with original copy
* The **advantage** of using CopyOnWriteArrayList over ArrayList is that, it doesn’t throws **ConcurrentModificationException** when **multiple threads** perform operation simultaneously

### CopyOnWriteArrayList v/s SynchronizedList :

* there is always a catch between **performance** and **thread-safety**
* choose wisely for your requirement

# **CopyOnWriteArrayList v/s ArrayList :**

|  |  |
| --- | --- |
| **CopyOnWriteArrayList** | **ArrayList** |
| CopyOnWriteArrayList is **synchronized** or newly introduced thread-safe class | ArrayList is **not** synchronized |
| For every **update** operation, a **new separate cloned copy**is created and there is **memory** & **merging overhead for JVM**  Hence**, performance** is relatively **low** when comparing with **ArrayList** | In multi-threaded environment, ArrayList is **faster** than **CopyOnWriteArrayList** as multiple threads **can operate**  Hence, **performance** is **high** as there is no need to acquire lock |
| While one thread **iterating CopyOnWriteArrayList** items, other threads happily can **modify**, as it **works** on separate cloned copy  And it **never** throws **ConcurrentModificationException** | While one thread **iterating** ArrayList items, if any other thread tries to **modify** same ArrayList object then **ConcurrentModificationException** is thrown |
| That’s it is **fail-safe iterator** | That’s it is **fail-fast iterator** |
| **Iterator of CopyOnWriteArrayList** can perform **read operation** safely; while iterating through **COWAL** items  But as soon as, **remove** operation is performed, compiler throws **UnsupportedOperationException** | **Iterator** of **ArrayList** can perform both **read** and **remove** operations; while iterating through **ArrayList** elements |
| Present in **java.util.concurrent** package and qualified class name is **java.util.concurrent** **.CopyOnWriteArrayList** | Present in **java.util** package and qualified class name is **java.util.ArrayList** |
| This is introduced in **Java 1.5** version | This is introduced in original collection framework in **Java 1.2** version |

### **When to use ArrayList ?**

* When there are **more number of retrievals** like accessing employee records against the employee code and
* Insertion and deletion is **very less** (or very minimal)
* **Reason:** internally when capacity exceeds then **new array with 50% more than original size** is created and older array data/items/elements are copied into new array
* Similarly, **lot** of **shifting** while deleting/removing an item/element from ArrayList
* But **if 2 or more threads** works on the **same ArrayList object simultaneously**; then compiler throws **ConcurrentModificationException**
* Use ArrayList instead of CopyOnWriteArrayList, when there is no thread-safety required

### **When to use CopyOnWriteArrayList ?**

* This is the **best suit to store elements as per insertion order** in a multi-threaded environment
* Where there are **more number of read** operation and **very less update**/modify operation
* Because for every update/modify operations, a **new separate cloned copy** is created
* And there is **overhead on JVM** to allocate **memory** & **merging** cloned copy with original copy
* The **advantage** of using CopyOnWriteArrayList over ArrayList is that, it doesn’t throws **ConcurrentModificationException** when **multiple threads** performes operation simultaneously

### ArrayList v/s CopyOnWriteArrayList :

* there is always a catch between **performance** and **thread-safety**
* Choose wisely for your requirement

# **Remove operation with CopyOnWriteArrayList**

In this article, we will discuss what happens when element is removed from **CopyOnWriteArrayList** and **ArrayList** while iterating using Iterator i.e.;

* **remove()** operation with **CopyOnWriteArrayList**; while iterating using **Iterator**
* **remove()** operation with **ArrayList**; while iterating using **Iterator**

# **1. CopyOnWriteArrayList:**

* Though, CopyOnWriteArrayList is very good choice over ArrayList while working in a **multi-threaded environment**
* but there are **certain limitation** too
* If we try to **remove** any element while iterating using Iterator;
* then **program fails** and **compiler** throws **ConcurrentModificationException**

#### **RemoveWhileIteratingCopyOnWriteArrayList.java**

[?](http://www.benchresources.net/copyonwritearraylist-with-remove-operation-in-java/#)

|  |
| --- |
| **package** in.bench.resources.concurrent.collection;    **import** java.util.Iterator;  **import** java.util.concurrent.CopyOnWriteArrayList;    **public** **class** RemoveWhileIteratingCopyOnWriteArrayList {    **public** **static** **void** main(String[] args) {    // creating CopyOnWriteArrayList of type String  CopyOnWriteArrayList<String> cowalStars =  **new** CopyOnWriteArrayList<>();    // adding elements to CopyOnWriteArrayList  cowalStars.add("Super Star");  cowalStars.add("Ultimate Star");  cowalStars.add("Rock Star");  cowalStars.add("Little Star");    System.out.println("Iterating using enhanced for-loop:\n");    // iterating CopyOnWriteArrayList using enhanced for-loop  **for**(String star : cowalStars) {  System.out.println(star);  }  System.out.println("\n1st: Iteration of COWAL "  + "completed... !!\n\n");    System.out.println("\n\n2nd: Iterating & removing:\n");    // iterating CopyOnWriteArrayList using Iterator  Iterator<String> itr = cowalStars.iterator();  **while**(itr.hasNext()){    String star = itr.next();    **if**(star.contains("Super")){  itr.remove();  }  **else**{  System.out.println(star);  }  }  }  } |

**Output:**

[?](http://www.benchresources.net/copyonwritearraylist-with-remove-operation-in-java/#)

|  |
| --- |
| Iterating using enhanced for-loop:    Super Star  Ultimate Star  Rock Star  Little Star    1st: Iteration of COWAL completed... !!          2nd: Iterating & removing:    Exception in thread "main" java.lang.UnsupportedOperationException  at java.util.concurrent.CopyOnWriteArrayList$COWIterator  .remove(CopyOnWriteArrayList.java:1040)  at in.bench.resources.concurrent.collection  .RemoveWhileIteratingCopyOnWriteArrayList  .main(RemoveWhileIteratingCopyOnWriteArrayList.java:36) |

**Explanation:**

* **1st time**, while **iterating** through **CopyOnWriteArrayList** contents using **enhanced for-loop**, all elements are printed to console as per **insertion order** (keep in mind, actually we **haven’t done anything** here like remove, etc)
* **2nd time**, when we tried to **remove** element after **comparison** while iterating using **Iterator** à program **fails** as **compiler**throws **UnsupportedOperationException**

### ArrayList v/s CopyOnWriteArrayList :

* But when same program is executed replacing **CopyOnWriteArrayList** by **ArrayList**,
* then program runs without any error
* there is no compile-time error

## **2. ArrayList:**

* It is very **safe to remove** elements from ArrayList while iterating using **Iterator**
* **compiler** **doesn’t** throws any runtime-exception like **UnsupportedOperationException**

#### **RemoveWhileIteratingArrayList.java**

[?](http://www.benchresources.net/copyonwritearraylist-with-remove-operation-in-java/#)

|  |
| --- |
| **package** in.bench.resources.concurrent.collection;    **import** java.util.ArrayList;  **import** java.util.Iterator;    **public** **class** RemoveWhileIteratingArrayList {    **public** **static** **void** main(String[] args) {    // creating ArrayList of type String  ArrayList<String> cowalStars = **new** ArrayList<>();    // adding elements to ArrayList  cowalStars.add("Super Star");  cowalStars.add("Ultimate Star");  cowalStars.add("Rock Star");  cowalStars.add("Little Star");    System.out.println("Iterating using enhanced for-loop:\n");    // iterating ArrayList using enhanced for-loop  **for**(String star : cowalStars) {  System.out.println(star);  }  System.out.println("\n1st: Iteration of AL completed... !!\n\n");    System.out.println("\n\n2nd: Iterating & removing:\n");    // iterating ArrayList using Iterator  Iterator<String> itr = cowalStars.iterator();  **while**(itr.hasNext()){    String star = itr.next();    **if**(star.contains("Super")){  itr.remove();  }  **else**{  System.out.println(star);  }  }  }  } |

**Output:**

[?](http://www.benchresources.net/copyonwritearraylist-with-remove-operation-in-java/#)

|  |
| --- |
| Iterating using enhanced for-loop:    Super Star  Ultimate Star  Rock Star  Little Star    1st: Iteration of AL completed... !!          2nd: Iterating & removing:    Ultimate Star  Rock Star  Little Star |

**Explanation:**

* **1st time**, while **iterating** through **ArrayList** contents using **enhanced for-loop**, all elements are printed to console as per **insertion order** (keep in mind, actually we **haven’t done anything** here like remove, etc)
* **2nd time**, when we tried to **remove** element after **comparison** while iterating using **Iterator** à program **doesn’t fails** and again all elements are printed as per **insertion order**
* But, **leaving out removed** element
* In this case, **Super star** because we have kept check for this only using **if-else condition**

# CopyOnWriteArrayList with Read and Update operations simultaneously

In this article, we will discuss how can we achieve **both read (iterate) and modify (remove/add) operations** simultaneously by 2 different threads using **CopyOnWriteArrayList** which isn’t possible with simple **ArrayList**

## **1. ArrayList:**

If **2 different threads** perform operations on same ArrayList object simultaneously, then compiler will throw **ConcurrentModificationException**

We will demonstrate a simple example using **ArrayList** which performs

* 1st thread **iterating** or **reading** element/objects one-by-one
* 2nd thread **removing** a particular element from List; while other thread is iterating ArrayList object

#### **IterateAndModifyArrayListSimultaneously.java**

[?](http://www.benchresources.net/copyonwritearraylist-with-read-and-update-operations-in-java/#)

|  |
| --- |
| **package** in.bench.resources.concurrent.collection;    **import** java.util.ArrayList;    // extending Thread class  **public** **class** IterateAndModifyArrayListSimultaneously **extends** Thread {    // creating ArrayList of type String  **static** ArrayList<String> alStars = **new** ArrayList<String>();    @Override  **public** **void** run() {    **try** {  // sleeping thread for 1000 ms  Thread.sleep(1000);    // removing element at 2nd position  String star = alStars.remove(1);  System.out.println("Thread 2: removed " + star);  }  **catch**(InterruptedException iex) {  iex.printStackTrace();  }  System.out.println("Removal is done... !!");  }    /\*\*  \* main() method  \* @param args  \* @throws InterruptedException  \*/  **public** **static** **void** main(String[] args) **throws** InterruptedException {    // adding elements to ArrayList  alStars.add("Rock Star");  alStars.add("Ultimate Star");  alStars.add("Little Star");    // creating another thread  Thread newThread = **new** Thread(  **new** IterateAndModifyArrayListSimultaneously());  newThread.start();    // iterating ArrayList using enhanced for-loop  **for**(String star : alStars) {    System.out.println("Thread 1 iterating : " + star);    // sleeping thread for 1500 ms, after every turn  Thread.sleep(1500);  }  System.out.println("Iterating AL completed... !!");  }  } |

**Output:**

[?](http://www.benchresources.net/copyonwritearraylist-with-read-and-update-operations-in-java/#)

|  |
| --- |
| Thread 1 iterating : Rock Star  Thread 2: removed Ultimate Star  Removal is done... !!  Exception in thread "main" java.util.ConcurrentModificationException  at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:819)  at java.util.ArrayList$Itr.next(ArrayList.java:791)  at in.bench.resources.concurrent.collection  .IterateAndModifyArrayListSimultaneously.main(  IterateAndModifyArrayListSimultaneously.java:46) |

**Explanation:**

* **Main thread** iterating ArrayList and **child thread** removing element at 2nd position (index-1) of same ArrayList object
* From output, it is clear that while **one thread is iterating** on ArrayList and **if any other thread** perform modify operation (i.e.; on the same ArrayList object, other thread is removing an element)
* Then compiler will throw **ConcurrentModificationException** i.e.; it is **fail-fast** iterator
* **Note:** sleep(ms) introduced to study demo example
* Because without sleep both thread will **execute independently** and complete its **execution** in **nano/pico seconds** and there won’t be any compile-time error
* Since we are trying to understand with **small amount** of **data,** where execution completes in nano seconds
* But with **large set of data**, introduction of sleep concept isn’t required
* As **execution time increases** for each thread, definitely **ConcurrentModificationException** is thrown

### Q) How to overcome above mentioned problem with ArrayList ?

* With **CopyOnWriteArrayList**, we can **overcome** this problem
* as it works on **different cloned copies** which is later merged into one/original copy by JVM

# **2. CopyOnWriteArrayList:**

When **2 different threads** performs operations on same **CopyOnWriteArrayList** object simultaneously, then compiler **won’t thrown** any runtime exception

This is the **advantage** of using **CopyOnWriteArrayList** over **ArrayList**

In the demo example,

* **1st thread iterates** through all elements of CopyOnWriteArrayList
* While **other thread can safely remove element at 2nd position** (i.e.; index-1)
* Compiler **doesn’t** throws any **ConcurrentModificationException** unlike ArrayList
* This is because, **CopyOnWriteArrayList** works on separate cloned copy and later JVM merges both original & cloned copies

#### **IterateAndModifyCopyOnWriteArrayListSimultaneously.java**

[?](http://www.benchresources.net/copyonwritearraylist-with-read-and-update-operations-in-java/#)

|  |
| --- |
| **package** in.bench.resources.concurrent.collection;    **import** java.util.concurrent.CopyOnWriteArrayList;    // implementing Runnable interface  **public** **class** IterateAndModifyCopyOnWriteArrayListSimultaneously  **implements** Runnable {    // creating CopyOnWriteArrayList of type String  **static** CopyOnWriteArrayList<String> cowalStars =  **new** CopyOnWriteArrayList<String>();    @Override  **public** **void** run() {    **try** {  // sleeping thread for 1000 ms  Thread.sleep(1000);    // removing element at 2nd position  String star = cowalStars.remove(1);  System.out.println("Thread 2: removed " + star);  }  **catch**(InterruptedException iex) {  iex.printStackTrace();  }  System.out.println("Removal is done... !!");  }    /\*\*  \* main() method  \* @param args  \* @throws InterruptedException  \*/  **public** **static** **void** main(String[] args) **throws** InterruptedException {    // adding elements to CopyOnWriteArrayList  cowalStars.add("Rock Star");  cowalStars.add("Ultimate Star");  cowalStars.add("Little Star");    // creating another thread  Thread newThread = **new** Thread(  **new** IterateAndModifyCopyOnWriteArrayListSimultaneously());  newThread.start();    // iterating CopyOnWriteArrayList using enhanced for-loop  **for**(String star : cowalStars) {    System.out.println("Thread 1 iterating : " + star);    // sleeping thread for 1500 ms, after every turn  Thread.sleep(1500);  }  System.out.println("Iterating COWAL completed... !!");  }  } |

**Output:**

[?](http://www.benchresources.net/copyonwritearraylist-with-read-and-update-operations-in-java/#)

|  |
| --- |
| Thread 1 iterating : Rock Star  Thread 2: removed Ultimate Star  Removal is done... !!  Thread 1 iterating : Ultimate Star  Thread 1 iterating : Little Star  Iterating COWAL completed... !! |

**Explanation:**

* When we executed same program **replacing** ArrayList with CopyOnWriteArrayList, then program **executed without any runtime** exception like **ConcurrentModificationException** i.e.; it is **fail-safe** iterator
* But there might be **different output** at **different execution point**
* Because, while **one thread iterating** on the object it can iterate **through all elements** or **updated items inside ArrayList** which is updated by 2nd thread
* In the above example, we have got **all elements** of **COWAL** while iterating; because **2nd** **thread’s update/**removal **doesn’t reflected to 1st thread’s** iteration/read
* **Same isn’t true with next iteration**, because next time there might be possibility of iterating through updated elements of COWAL (in this case 1st thread got updated values from 2nd thread)

### Lets us print other possibility as well

**Output:**

[?](http://www.benchresources.net/copyonwritearraylist-with-read-and-update-operations-in-java/#)

|  |
| --- |
| Thread 2: removed Ultimate Star  Removal is done... !!  Thread 1 iterating : Rock Star  Thread 1 iterating : Little Star  Iterating COWAL completed... !! |

From above output, it is clear that **1st thread got updation from 2nd thread** and 1st thread iterated only on updated elements of COWAL

**java.util.Stack<E>**

**public class Stack<E> extends Vector<E>**

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example: Deque<Integer> stack = new ArrayDeque<Integer>();

### Java Stack Methods

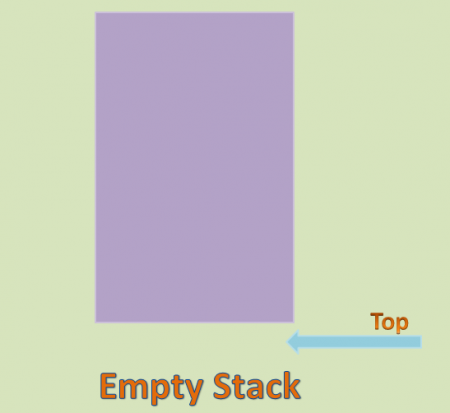
Java Stack extends Vector class with the following five operations only.

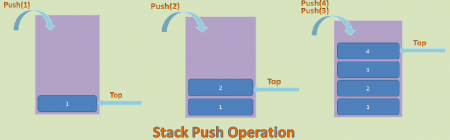
1. boolean empty(): Tests if this stack is empty.
2. E peek(): Looks at the object at the top of this stack without removing it from the stack.
3. E pop() : Removes the object at the top of this stack and returns that object as the value of this function.
4. E push(E item) : Pushes an item onto the top of this stack.
5. int search(Object o) : Returns the 1-based position where an object is on this stack.

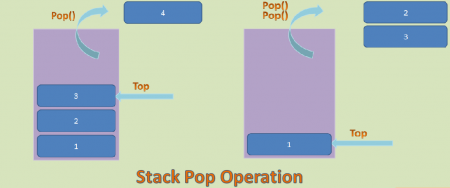
### How Stack’s push() and pop() operations works Internally?

As we know Stack’s push() and pop() are most frequently used Stack operations. The push() operation is used to insert an element into a Stack at the top. The pop() operation is used to remove the top element from a Stack.

Stack data structure has one internal property: top to refer top element of that stack. If Stack is empty, this top refers to the before first element as shown below:



As shown in the below diagram, Stack’s Push operation always inserts new element at the top of the Stack.  


As shown in the below diagram, Stack’s Pop operation always removes an element from the top of the Stack.  


Stack’s isEmpty() operation returns True if there are no elements, otherwise, it returns False.

**java.util.Vector<E>**

**public class Vector<E>extends AbstractList<E>implements List<E>, RandomAccess, Cloneable, Serializable**

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement. The capacity is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of capacityIncrement. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

The iterators returned by this class's iterator and listIterator methods are fail-fast: if the vector 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. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The Enumerations returned by the elements method are not fail-fast.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

As of the Java 2 platform v1.2, this class was retrofitted to implement the List interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.

The Vector class implements a growable array of objects. Vectors basically falls in legacy classes but now it is fully compatible with collections.

* Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
* They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
* It extends **AbstractList** and implements **List** interfaces.

**Constructor:**

* **Vector()**: Creates a default vector of initial capacity is 10.
* **Vector(int size):** Creates a vector whose initial capacity is specified by size.
* **Vector(int size, int incr):** Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time that a vector is resized upward.
* **Vector(Collection c):** Creates a vector that contains the elements of collection c.

**Important points regarding Increment of vector capacity:**   
If increment is specified, Vector will expand according to it in each allocation cycle but if increment is not specified then vector’s capacity get doubled in each allocation cycle. Vector defines three protected data member:

* **int capacityIncreament:** Contains the increment value.
* **int elementCount:** Number of elements currently in vector stored in it.
* **Object elementData[]:** Array that holds the vector is stored in it.

# Difference between ArrayList and Vector

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

However, 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 the number of elements exceeds from its capacity. | Vector **increments 100%** means doubles the array size if the total number of elements 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 a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object. |
| 5) ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |

