**Collection:** A group of objects which stores homogenous and heterogenous elements.

**Why collection:** Collection is used for storing multiple objects as a single group and sending all objects at a time from one class to another class as a method parameter argument and also as a method return type name.

**Different ways to store values in Java JVM:**

1)using Variable: stores single value.

2)using class object: can store multiple fixed number of values of different type.

3)using Array object: can store multiple fixed number of values of same type

4)using collection object: when you don’t know the data type and size of sending the values by the user and if you want the store all the values as a single object and wanted to send all these values from one application to another application with single method parameter.

**Problems with Array:**

1)Fixed size. Once we create an array there is no chance of increasing or decreasing the size based on our requirement.

2)Homogenous elements(similar):

3)No underlying data strucure(i.e. there are no readymade methods like sorting). Arrays is not implemented on some standard data structure hence readymade methods are not available. Hence for every requirement we need to write the code explicitly which increases complexity of programming.

To over come above problems of arrays we should go for Collections.

**Collections:**

* collections are growable in Nature i.e. based on our requirement we can increase or decrease the size.
* Collections can hold both the homogenous and heterogenous elements.
* Every collection class is implemented based on some standard Data structure. Hence for every requirement ready made method is available. Being a programmer we are responsibe to use and we are not responsible to implement these methods.

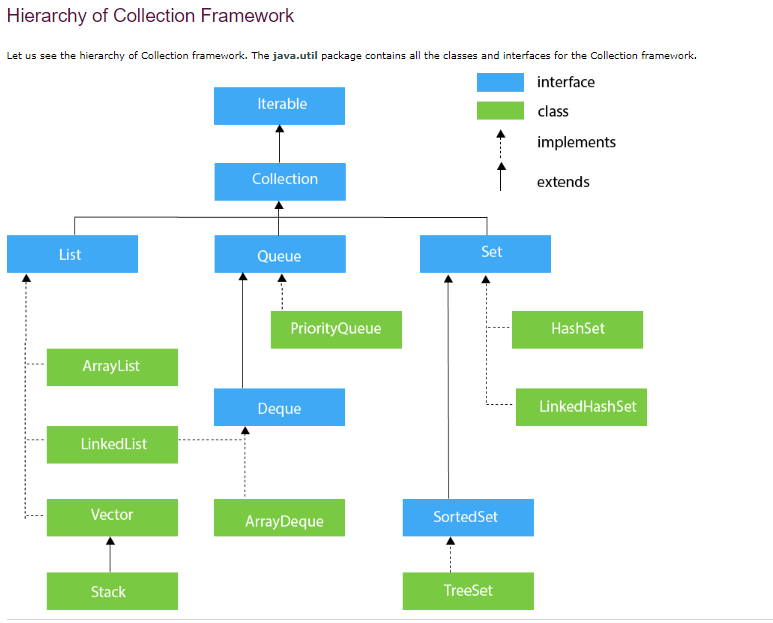
**Difference between Array and Collections:**

|  |  |
| --- | --- |
| **Array** | **Collections** |
| Fixed in size | Growable in Nature. |
| With respect to memory arrays are not recommended to use | With respect to memory collections are recommended to use |
| With respect to performance arrays are recommended to use | With respect to performance collections are recommended to use |
| Hold homogenous elements | Homogenous and heterogenous elements |
| No underlying Data structure | Implemented based on standard data structure. |
| Can hold both primitives and objects | Can hold only object types but not primitives |

**Difference between Collection and Collections in Java:**

**Collection is a top level interface of Java collection of java collection framework whereas collections is an utility class.**

1. **Collection** is an interface and Collections is a class.
2. Both are belongs to Java.Util package
3. **Collection** is a base interface for set, link and queue.
4. **Collections** is a class and it is called utility class
5. **Collections** is an utility class contains some predefined methods so that we can use while working with collection type of classes(treeset, arraylist, linkedlist etc.)
6. **Collection** is a root level interface of the Java Collection Framework. Most of the classes in Java Collection Framework inherit from this interface. List, Set and Queue are main sub interfaces of this interface.
7. **Collections** is an utility class in java.util package. It consists of only static methods which are used to operate on objects of type Collection. For example, it has the method to find the maximum element in a collection, it has the method to sort the collection, it has the method to search for a particular element in a collection.
8. JDK doesn’t provide any direct implementations of this interface. But, JDK provides direct implementations of it’s sub interfaces. ArrayList, Vector, HashSet, LinkedHashSet, PriorityQueue are some indirect implementations of Collection interface. Map interface, which is also a part of java collection framework, doesn’t inherit from Collection interface. Collection interface is a member of java.util package.



**9 Key interfaces in collection framework:**

1. Collection
2. List
3. Set
4. SortedSet
5. Navigableset
6. Queue
7. Map
8. SortedMap
9. NaviagableMap

**1.Collection(Interface):**

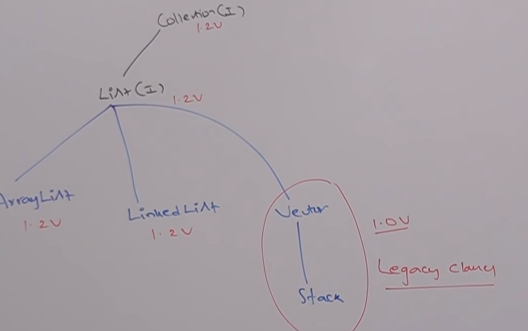
* If we want to represent a group of individual objects as a single entity then we should go for collection.
* Collection interface defines the most common methods which are applicable for any collection object.
* In general, collection interface is considered as root interface of collection framework.
* There is no concrete class which implements collection interface directly.

**List(Interface):**

* It is the child interface of Collection.
* If we want to represent a group of individual objects as a single entity **where duplicates are allowed and insertion order must be preserved** then we should go for List.

**Implementation classes of List interface are ArrayList, LinkedList, Vector and Stack**

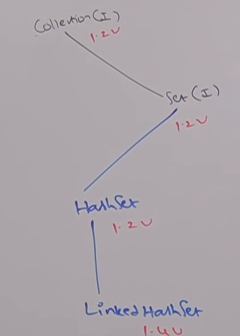
In 1.2 Version Vector and stack classes are modified/updated to implement list interface.



**Set(Interface):**

* Set is the child interface of collection.
* Duplicates values are not allowed and the insertion order is not preserved

**Implementation classes of Set interface are HashSet and LinkedHashSet**



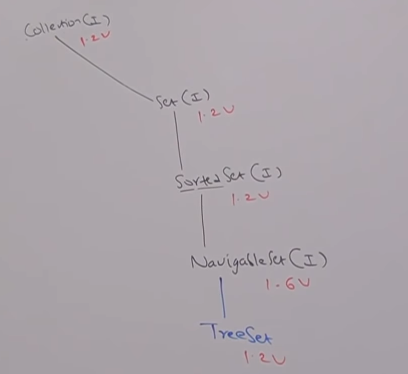
**SortedSet(Interface):**

* It is the child interface of Set.
* If we want to represent a group of individual objects as a single entity **where duplicates are not allowed and all objects should be inserted according to some sorting order** then we should go for SortedSet.

**NavigableSet(Interface):**

* It is the child interface of SortedSet.
* It contains Several methods for Navigation purposes.

**Implementation class of NavigableSet is TreeSet**



**What is the difference between List and Set:**

|  |  |  |
| --- | --- | --- |
| **SLNO** | **List** | **Set** |
| 1 | Duplicates are allowed | Duplicates are not allowed |
| 2 | Insertion order preserved | Insertion Order not preserved |
| 3 | Implementation classes are **ArrayList, LinkedList, Vector and Stack** | **HashSet and LinkedHashSet** |

**Queue(Interface):**

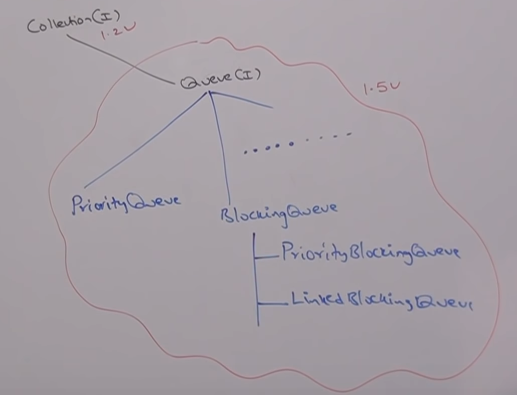
Queue is child interface of collection.

If we want to represent a group of individual objects **prior to processing** then we should go for Queue.

Usually, Queue follows First In First Order(FIFO) but based on our requirement we can implement our own prior order also.

Example: Before sending an email all mail id’s we have to store in some DataStructure. In which order we added mail ids in the same order only mail should be delivered.For this requirement, Queue is best choice.

Implementation classes of Queue are PriorityQueue and BlockingQueue. For BlockingQueue there are two classes those are PriorityBlockingQueue and LinkedBlockingQueue.



**Note:** All the above interfaces(Collection, List, Set, SortedSet, NavigableSet and Queue) meant for representing a group of individual objects.

If we want to represent a group of objects as Key-Value pairs then we should go for Map.

**Map(Interface):**

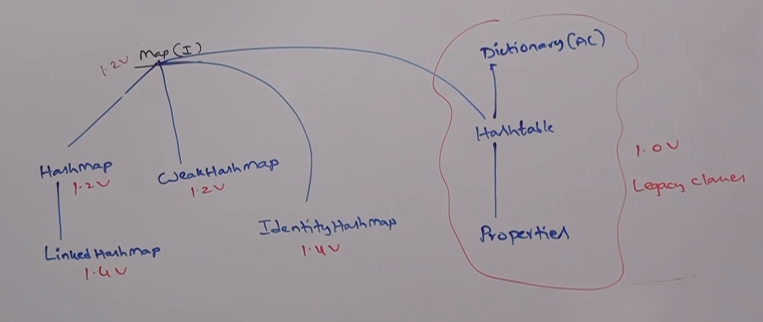
Map is not child interface of Collection.

If we want to represent a group of objects as Key-Value pairs then we should go for Map.

Both Key and Value are Objects only

Duplicate keys are not allowed but values can be duplicated.

Implementation classes of Map are HashMap, LinkedHashMap, WeakHashMap, IdentityHashMap, Hashtable, Dictionary and Properties(properties is the class and Dictionary is the abstract class for Hashtable)



**SortedMap(Interface):**

It is the child interface of Map interface.

If we want to represent a group of **key value pairs according to some sorting order of keys** then we should go for SortedMap.

In SortedMap the sorting should be based on Key but not based on value.

**NavigableMap(Interface):**

It is the child interface of SortedMap interface.

It defines several methods for Navigation purposes.

Implementation class is Tree Map.

**Note:**

The following are legacy characters present in collection framework.

Enumeration (Interface)

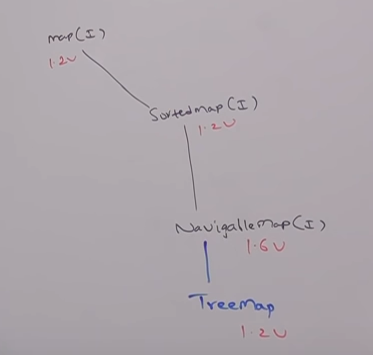
Dictionary (Abstract Class)

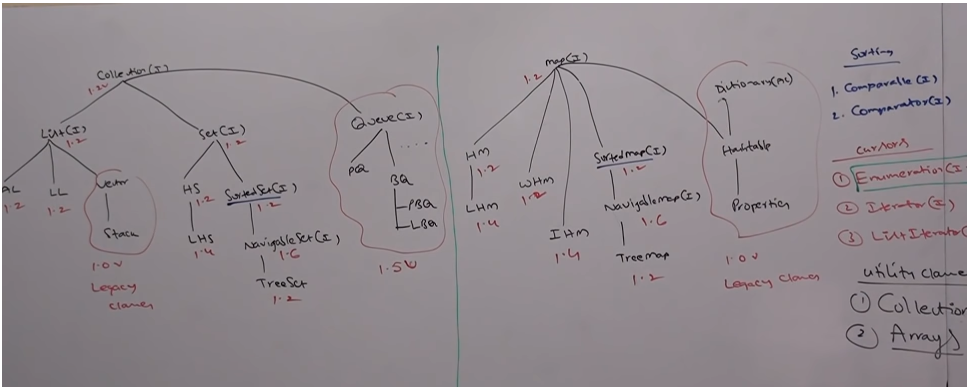
Vector (class)

Stack (Class)

Hasttable (Class)

Properties (Class)





**Methods in Collection Interface:**

Boolean Add(Object o)

Boolean AddAll(Collection c)

Boolean Remove(Object o)

Boolean removeAll(collection c) – To remove a group of objects

Void clear() – to remove all objects

Boolean retainAll(Collection c) – To remove all objects except those present in collection c

Boolean Contains(Object o)

Boolean ContainsAll(collection c)

Boolean isEmpty()

Int size()

Object[] toArray() convert the collection to array. Returns the object array.

Object[] toArray()

Iterator iterator() – iterates through the collection. Returns the iterator object

Iterator iterator().

Note: There is no concrete class which implements collection interface directly.

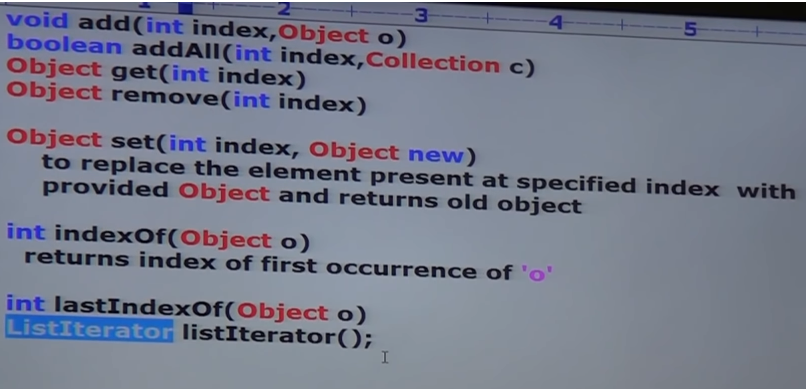
**List(Interface):**

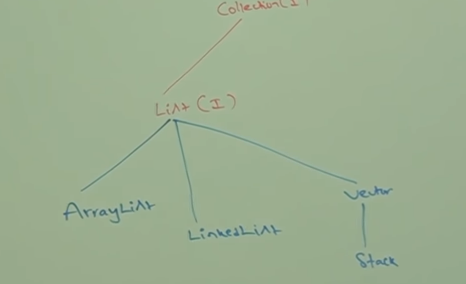
List is child interface of collection.

If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order must be preserved then we should go for List.

We can preserve insertion order with index and we can differentiate duplicate objects by using index. Hence Index will play very important role in List.

List interface defines the following specific methods.





**ArrayList:**

1. The underline datastructure is resizable array or growable array.
2. Duplicates are allowed.
3. Insertion order is preserved.
4. Heterogenous objects are allowed.(Except Treeset and TreeMap everywhere heterogenous objects are allowed)
5. Null insertion is possible.

Constructors of ArrayList:

1)ArrayList l = new ArrayList() creates an empty arraylist object with default initial capacity 10.

Once arraylist reaches its max capacity then a new arraylsit object will be created with

new capacity =(currentcapacity\*3/2)+1

2)ArrayList l=new ArrayList(int initialcapacity) creates an empty arraylist object with specified initial capacity.

3)**ArrayList l=new ArrayList(Collecton c)** creates an equivalent arraylist object for the given collection.

**Example of ArrayList:**

**package** collections;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**public** **class** List {

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

ArrayList<Object> al = **new** ArrayList<Object>();

al.add("Narayana");

al.add(10);

al.add(**null**);

System.***out***.println(al);

**//Method 1: Iterating through ArrayList**

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

System.***out***.print(al.get(i) + " ");

}

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

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

// Creating a list

ArrayList<Integer> l1 = **new** ArrayList<Integer>();

l1.add(0, 1); // adds 1 at 0 index

l1.add(1, 2); // adds 2 at 1 index

System.***out***.println(l1);

**int** l1size=l1.size();

System.***out***.println("size of l1 list is "+l1size);

**//Method 2: For Each Loop for iterating ArrayList**

**for**(Integer i: l1) {

System.***out***.print(i + " ");

}

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

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

ArrayList<Integer> ai = **new** ArrayList<Integer>();

ai.add(10);

ai.add(15);

ai.add(10);

l1.addAll(1, ai);

System.***out***.println(ai);

**// Method3: Looping ArrayList using Iterator**

Iterator it=ai.iterator();

**while**(it.hasNext()) {

System.***out***.print(it.next() + " ");

}

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

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

System.***out***.println(l1);

**//Method 4: Using For Each Method of Java 8**

l1.forEach(number->System.***out***.print(number + " "));

}

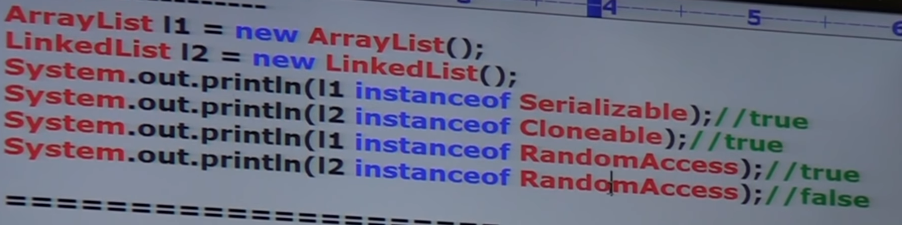
}

**Important Points on ArrayList:**

1. Usually we can use collections to hold and transfer object from one location to another location(container). To provide support for this requirement every collection class by default implements serializable and clonable interfaces.
2. ArrayList and Vector classes implements RandomAccess interface so that any random element we can access with the same speed.

**RandomAccess:**

RandomAccess interface present in Java.util package and it doesn’t contain any methods. It is a Marker interface, where required ability will be provided automatically by the JVM.



* Arraylist is the best choice if ever frequent operation is retreival operation because ArrayList implements RandomAccess interface.
* ArrayList is the worst choic if ever frequent operation is insertion or deletion in the middle.

**Difference between ArrayLIst and Vector:**

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| Every method present in the arraylist is non synchronized. | Every method present in the vector is synchronized |
| At a time, multiple threads are allowed to operate on ArrayList object and hence it is not thread-safe | At a time, only one thread is allow to operate on vector object and hence it is thread safe |
| Relatively performance is high because threads are not required to wait to operate on Arraylist object | Relatively performance is low because threads are required to wait to operate on vector object. |
| Introduced in 1.2V and it is non leagcy | Introduced in 1.0V and it is legacy |

**How to get synchronized version of ArrayList object:**

By default ArrayLIst is non synchronized but we can get synchronized version of ArrayList by using <SynchronizedList> method of collections class.

Public static List synchronizedList(List l)

ArrayList l = new ArrayList();

List l1=collections. synchronizedList(l);//here l1 is synchronized list and l is non synchronized.

Similarly we can get synchronized version of set and map objects by using the following methods of collections class.

Public static set synchronizedSet(set s)

Public static Map synchronizedMap(Map m)

**LinkedList:**

* The underline datastructure is doublelinkedlist.
* Insertion order is preserved.
* Duplicate objects are allowed.
* Heterogenous objects allowed.
* Null insertion is possible.
* It implements serializable and clonable interface but not randomaccess.
* Linked list is the best choice if ever the frequent operation is insertion or deletion in the middle..
* It is the worst choice if ever the frequent operation is retreival operation.

**Constructor of LinkedList:**

LinkedLIst l = new LinkedList(); creates an empty linked list object

LinkedList l=new LinkedList(collection c) – creates an equivalent linkedlist object for the given collection.

**LinkedList class specific methods:**

Usually we can use linked list to develop stacks and queues. To provide support for this requirement linkedlist class defines the following specific methods.

Void addFirst(Object o)

Void addLast(Object o)

Object getFirst()

Object getLast()

Object removeFirst()

Object removeLast()

**Differences between ArrayList and LinkedList:**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| ArrayList is the best choice if ever the frequent operation is retreival operation | LinkedList is the best choice ifever frequent operation is insertion or deletion in the middle |
| ArrayList the worst choice if ever the frequent operation is insertion or deletion in the middle because internally several shift operations are performed | LinkedList is the worst choice if ever frequent operation is retreival operation |
| In ArrayList the elements will be stored in consecutive memory locations and hence retreival operation will bcome easy | In LinkedList, the elements wont be stored in consecutive memory locations and hence retrieval operations will become difficult or complex. |

**Vector:**

The underline datastructure is resizable array or growable array.

Insertion order is preserved.

Duplicates are allowed.

Heterogenous objects are allowed.

Null insertion is possible.

It implements serializable, clonable and random access interfaces.

Every method present in the vector is synchronised and hence vector object is thread safe.

**Constructors:**

**1)Vector V = new Vector()** creates an empty object with default initial capacity 10.

Once vector reaches its max capacity then a new vector object created with new capacity = current capacity \* 2

**2)Vector V = new Vector(int initial capacity)** creates an empty vector with specified initial capacity.

**3)Vector V = new Vector(int initial capacity,int incremental capacity)**

**4)Vector V = new Vector(Collection c)** creates an equivalent vector object for the given collection. This constructor meant for interconversion between collection objects.

**Vector Specific Methods:**

**Add objects:**

add(Obejct o)

addIndex(int index, object o)

addElement(Object o)

**To remove objects:**

Remove(Object o)

removeELement(Object o)

removeElementAt(int index)

removeAllElements()

**To get Objects:**

Object elementAt(int index)

Object firstElement()

Object lastElement()

**Other methods:**

Int size()

Int capacity()

Enumeration elements()

**Cursors of Java:** If we want to get objects one by one from the collection then we should go for cursor.

There are 3 types of cursors available in java.

1)Enumeration

2)iterator

3)listiterator

**1)Enumeration:** We can use enumeration to get objects one by one from legacy collection object

We can create enumeration object using elements method of vector class.

Public enumeration elements();

Eg: Enumeration e = new v.elements(); //v is vector object

**Methods:**

Public boolean hasMoreElements()

Public object nextElement()

