* **Collections Framework - the need**
* As number of variables increase in a program, it is impossible to declare and use them as individual variables. This brings down the readability of the code and makes it verbose.
* To tackle the increased size of the individual variables, we go for Arrays. Arrays provide ways to group individual variables and reference them using a single variable.
* But Arrays pose a lot of limitations. Below are some of them:
* Arrays are fixed in size. This is not helpful when we don’t know what will be the size of the variables that we are going to deal with.
* Arrays are homogenous in nature. This means that if we declare an integer array, we can’t use the same to add a float number or a double number.
* Arrays are not implemented on any underlying data structure. Due to this, we do not get ready made method support. It will be the sole responsibility of the programmer to write custom logic for the arrays.
* Collections overcome the problem of the array. Below are the advantages of the collections:
* Collections are growable in size.
* Collections are both homogenous and heterogenous.
* Every collection class are developed based on some standard data structure. Hence we get ready made method support for all the classes in the collections framework.
* **Difference between Arrays and Collections**

| **Parameter** | **Arrays** | **Collection** |
| --- | --- | --- |
| Size | fixed | growable |
| Memory | not recommended | recommended |
| Performance | better | not upto the mark |
| Data types | only homogenous | both homogenous and heterogenous |
| method support | not available | readily available |
| objects holding | primitives + objects | objects only |

* **Collection and Collections framework**
* Collection is a group of individual objects as a single entity.
* Collection framework is the group of several classes and interfaces which can be used to group several objects are a single entity.
* **Key interfaces of the collection framework**
* Interfaces provide more information compared to class.
* There are 9 key interfaces in the collection framework. They are
* **Collection** - Use this when we want to represent a group of individual objects as a single entity. This interface defines the most common objects required to operate on collection. Ex. add(), contains(), isEmpty() etc

**There is no concrete class that implements Collection interface directly.**

* **List** - This interface makes a collection have duplicates and keeps track of insertion order.

List interface is the child of Collection interface.

**List has 3 implementation 1. ArrayList 2. LinkedList 3. Vector 4. Stack**

Vector and stack are called legacy classes.

* **Set -** This interface makes a collection to have unique elements and it doesn’t care about insertion order.

Set is child interface of Collection interface.

**Set has 2 implementation 1. HashSet and its child 2. LinkedHashSet**

* **SortedSet -** SortedSet is a child interface of Set.

This interface makes sure all elements are unique and the elements are inserted in some sorting order.

* **NavigableSet -** NavigableSet is the child interface of SortedSet.

This interface defines methods for easy navigation of the set.

**TreeSet is the implementation class of the NavigableSet.**

* **Queue** - Use Queue when you want to represent a group of objects before processing.

Queue is the child interface of collection.

**Queue has 4 implementation classes 1. PriorityQueue 2. BlockingQueue**

**3. LinkedBlockingQueue and 4. PriorityBlockingQueue**

**LinkedBlockingQueue and PriorityBlockingQueue are child classes of**

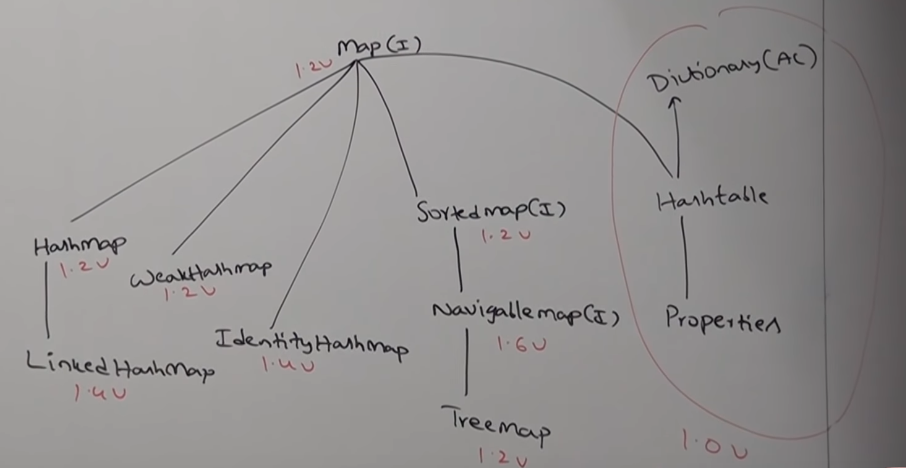
**BlockingQueue.**

* **Map - This is not a child interface of the Collection interface**

Map is a data struction having K,V pairs.

keys and values are objects.

Values can be duplicated but Keys are always unique



* **SortedMap** - Representing a group of object as K,V pairs in some sorting order, we use SortedMap.

**SortedMap is the child interface of the map**

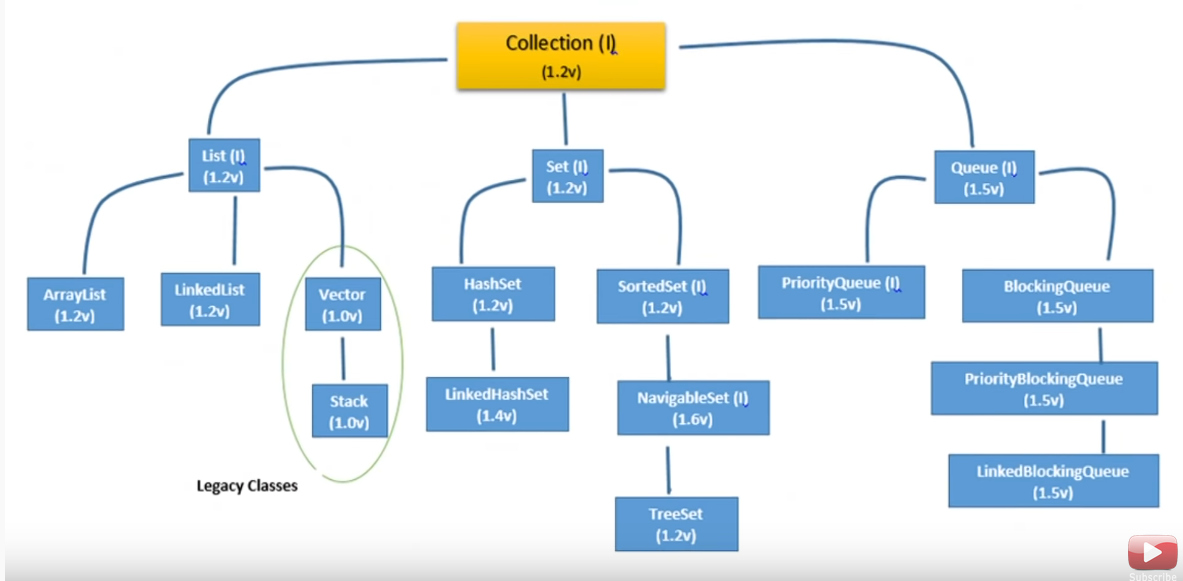
* **NavigableMap** - **This is the child interface of SortedMap**

NavigableMap defines several utility methods that makes navigation on the map easy.

**TreeMap is the implementation of NavigableMap**

* **Difference between Collection and Collections**
* **Collection is an interface and Collections is a class.**
* Collections is a utility class that defines implementations for utility methods defined in the Collection interface.
* **Difference between List and Set**

| **List** | **Set** |
| --- | --- |
| Duplicates are allowed | duplicates are not allowed |
| insertion order preserved | Order is not preserved. |

* **Complete overview of Collection Framework**
* 
* There are 2 areas where we are talking about sorting: SortedSet and SortedMap
* If we want default/natural sorting order, we must go for **Comparable** interface.
* If we want custom sorting, we must go for **Comparator** interface.
* When we want to fetch the objects of a collection one by one, we must use **Cursors.**
* There are 3 interfaces that define the Cursors in Java: **1. Enumeration 2. Iterator 3. ListIterator**
* There are 2 utility classes in Collection frameworks: **1. Collections 2. Arrays**
* **Collection Interface**
* Use Collection when we want to represent a group of individual object as a single entity.
* Collection interface defines the most common methods applicable for a collection object.
* Collection interface is considered as the root interface of the Collection framework.
* Following is the list of methods and their functionalities

| add( Object o ) | adds object o into the collection |
| --- | --- |
| addAll( Collection c ) | adds collection c into a collection |
| remove( Object o ) | removes object o from the collection |
| removeAll( Collection c ) | removes collection c from the collection |
| clear() | removes all elements from collection |
| retainAll( Collection c ) | removes all element except collection c |
| isEmpty() | returns true if collection is empty a nd false otherwise |
| size() | returns size of the collection |
| contains( Object o ) | returns true if collection contains object o and false otherwise |
| containsAll( Collection c ) | returns true if the collection contains collection c false otherwise |
| Object[] a = c.toArray() | converts the collection c into an array |
| Iterator i = c.iterator() | returns iterator for the collection. |

* Collection interface doesn’t contain any method to retrieve an object. There is no concrete class that implements the Collection Interface directly.
* **List Interface**
* List is the child interface of collection.
* Go for List if we need to preserve insertion order and duplicates are allowed.
* Duplicates in a List can be differentiated using indices.
* Insertion order is preserved using index.
* Since List is the child interface of Collection, all methods of Collection interface is available to List interface.
* Index in the List is 0 based
* Following are the methods specific to the List Interface

| add( int i, Object o ) | adds o at ith index |
| --- | --- |
| addAll( int i, Collection c ) | adds collection c from index i |
| get( int i ) | gets element at ith index |
| remove ( int i ) | removes element at ith index |
| set ( int i, Object o ) | sets object o at ith index |
| indexOf( Object o) | gets the first occurence of o in the list |
| lastIndexOf( Object o ) | gets the last occurence of o in the list |
| ListIterator li = list.listIterator() | gets the list iterator object to the specified list |

* **ArrayList class**
* ArrayList is based on resizable array or growable array. That is the underlying data structure.
* Duplication is allowed. Insertion order is preserved.
* We can insert heterogenous objects into ArrayList.
* In entire collection framework, heterogenous objects can be stored in all collections except TreeSet and TreeMap. TreeSet and TreeMap store in some sorting order and to sort, comparision must happen. Only homogenous objects can be compared.
* Null insertion is possible.
* To send objects across a network, they must be serializable. Hence all collection objects implement Serializable and Clonable interfaces.
* ArrayList and Vector implement RandomAccess interface. By doing so, any random element within them can be accessed with the same speed.
* If the frequent operation is retrival, highly recommended to go for ArrayList.
* ArrayList is not recommended if frequent operation is insertion in the middle or removal from the middle.
* RandomAccess is a marker interface. It does not contain any methods.
* **Constructors for ArrayList class**
* **ArrayList k = new ArrayList();**

creates an empty ArrayList with size 10. Once limit is reached, the ArrayList resizes according to formula (CurrentSize \* 1.5) + 1

* **ArrayList k = new ArrayList( int initialCapacity);**

creates empty ArrayList with size initialCapacity.

* **ArrayList k = new ArrayList( Collection x );**

if we want to create the equivalent ArrayList object for collection x, we use the above constructor.

* **Additional details about collections.**
* Collections are used to transfer data from one place to another. To do this on a network, every collection object must be serializable. All collection classes implement the Serializable interface.
* All collection object can be cloned into duplicates. Hence all collection classes implement Cloneable interface.
* ArrayList and Vector class implement RandomAccess interface. This makes the classes access any object with the same speed.
* ArrayList is the best choice if the frequent operation is retrieval.
* ArrayList is a worst choice if the frequent operation is insertion or deletion.
* **Differences between ArrayList and Vector**

| **ArrayList** | **Vector** |
| --- | --- |
| Methods of the class is not synchronised. | Methods are synchronised. |
| Thread safety is not there. Multiple threads are allowed to operate at a time | Thread safety is there. Only one thread can operate at a time |
| Improved performance. Quick and faster performance | Poorer performance. Slow and poor performance. |

* **Getting thread safe ArrayList object**
* ArrayList l1 = new ArrayList(); => Non synchronised
* List l = Collections.synchronizedList(l1); => synchronized method
* The method signature => public static List synchronizedList( List l )
* Similarly we can get synchronised version of map and set
* public static Set synchronizedSet( Set s )
* public static Map synchronizedMap( Set m )
* **LinkedList details**
* LinkedList is helpful when we need to insert or delete frequently in between elements.
* Elements in LinkedList wont be stored in consecutive memory location. Nodes are connected to each other via pointers.
* Searching elements in LinkedList is poor in performance as it must start from first element only.
* Underlying data structure for LinkedList is doubly linked list.
* Insertion order is preserved in LinkedList
* Duplicates are allowed.
* Heterogenous objects are allowed.
* Null insertion is allowed.
* LinkedList implements Serializable and Cloneable Interface.
* LinkedList is used to develop stack and queue. Hence 6 methods are available for this purpose.
* addFirst(Object o)
* addLast(Object o)
* getFirst()
* getLast()
* removeFirst()
* removeLast()
* **Constructors in LinkedList**
* **LinkedList k = new LinkedList();** creates an empty linked list.
* LinkedList does not follow any concept of reserving a memory location etc etc. We cre ate node and attach.
* **LinkedList k = new LinkedList(Collection c);** creates an equivaled linked list object for any collection object
* Differences between ArrayList and LinkedList

| **ArrayList** | **LinkedList** |
| --- | --- |
| Best choice if frequent operation is retrival. This is because ArrayList implements RandomAccess. | Best choice if our frequent operation is insertion or deletion in the middle. |
| Worst choice if frequent operation is insertion or deletion in the middle | Worst choice if our frequent operation is retrival. |
| ArrayList underlying data structure is resizable or growable array | LinkedList underlying data structure in doubly linked list. |
| ArrayList implements RandomAccess Interface | LinkedList doesnt implement RandomAccess interface. |

* **Vector class details**
* Underlying data structure is resizable/growable array.
* Duplicates are allowed.
* Insertion order preserved.
* Null insertion possible
* Heterogenous objects are allowed.
* Implements Serializable and Cloneable interface.
* Implements RandomAccess interface
* Most of the methods in Vector class is synchronised. Therefore Vector object is always thread safe.
* Vector is best choice if we need to perform retrival operation.
* **Constructors in Vector class**
* **Vector v = new Vector();**

creates empty vector object with initial capacity = 10; new capacity = (current capacity \* 2)

* **Vector v = new Vector(int initialSize);**

creates vector of known capacity. Improved performance

* **Vector v = new Vector(int initialSize,int incrementalCapacity);**

creates vector objectof initialSize. If we add new elements, increments size by incrementalCapacity.

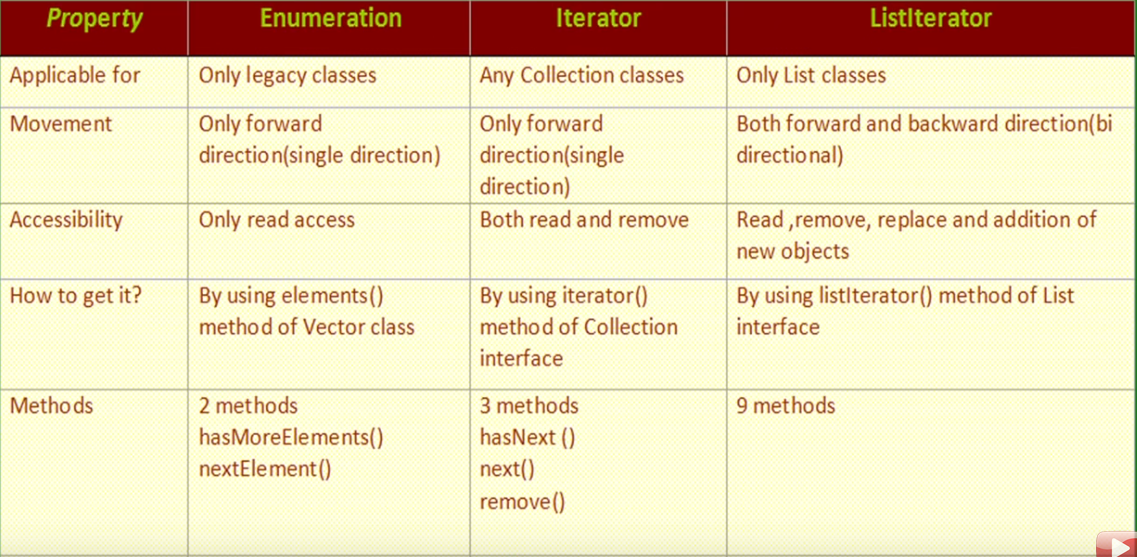
* **Vector v = new Vector( Collection c );**

creates equivalent vector object for collection object c

* **Stack**
* Stack is the child class of vector.
* Specially designed class for LIFO
* **Stack s = new Stack();** has only one constructor
* **Methods in stack class**
* push( Object o )
* pop()
* peek() => to return the top element of stack without removing
* empty() => to check whether stack is empty or not
* search(object) => returns offset. offset is the position where element is found.
* **Cursors in Java**
* Cursors are object used to retrive objects from a collection one by one.
* There are 3 cursors.
* Enumeration
* Iterator
* ListIterator
* **Enumeration**
* We can get enumeration by using the elements() method on the vector object

Enumeration e = v.elements()

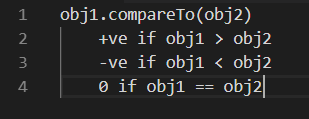
* hasMoreElements(); returns true if there are more methods
* nextElement(); returns next element in the vector
* **Limitations of Enumeration**
* Enumeration is a legacy cursor. Hence it can be applied to only legacy classes.
* By using enumeration, we can only get read access. We cant perform remove operation.
* **Iterator**
* Iterator is applicable for all collection classes.
* Iterator is a universal iterator.
* Iterator helps us do read and remove operation.
* **Iterator itr = c.iterator();** c is any collection object
* **hasNext();** returns whether next element is there or not
* **next();** returns next object
* **remove();** removes the current object
* **Limitations of Iterator**
* Enumeration and Iterator always moves in the forward direction. They are single direction cursors.
* Iterator cant perform addition of new objects into the collection.
* **ListIterator**
* Most powerful cursor. It is a bidirectional cursor.
* We can perform read, remove, replacement and addition of new objects.
* ListIterator is the child interface of the iterator class.
* **ListIterator k = c.listIterator();** c is any list(AL, LL, V) object
* **Methods in ListIterator**
* **hasNext();** returns whether next element is there or not
* **next();** returns next object
* **nextIndex();** returns index of the next element
* **hasPrevious();** returns whether previous element is there or not
* **previous();** returns previous object
* **previousIndex();** returns index of the previous element
* **remove();** removes the current object
* **set(Object new);** sets new object at current location
* **add(Object new);** adds new object at current location
* **ListIterator limitation**
* Only applicable to List interface.
* **Differences between the cursors**



* **HashSet**
* Underlying data structure is HashTable
* Duplicates are not allowed. No complile time error or a run time exception if we try to add duplicates. add() method will return false.
* Insertion order is not preserved. All objects will be inserted based on the hashcode.
* Heterogenous object allowed
* Null insertion allowed
* Implements Serializable and Cloneable interface.
* If frequent operation is search, HashSet is the best choice.
* **Constructors for the HashSet class**
* **HashSet s = new HashSet();** empty HashSet with default initial capacity = 16 and fillRatio = 0.75
* **HashSet s = new HashSet(initialCapacity);** created HashSet with specified initialCapacity and fillRatio = 0.75
* **HashSet s = new HashSet(initialCapacity, loadFactor)**
* **HashSet s = new HashSet(Collection c)**
* **LinkedHashSet**
* Child class of HashSet.
* Use LinkedHashSet when we want unique elements and insertion order preserved.

| **HashSet** | **LinkedHashSet** |
| --- | --- |
| underlying data structure is HashTable | Underlying data structure is hybrid combination of linked list and HashTable |
| Insertion order is not preserved | insertion order is preserved |

* **SortedSet**
* Used to represent a group of objects according to some sorting order without any duplicates, use SortedSet
* SortedSet, in contrast to Set, we have 6 methods to get the sorting and ordering
* **first();** returns first element
* **last();** returns last element
* **headSet(Object o);** returns set of objects less than o
* **tailSet(Object o);** returns set of objects greater than o
* **subSet(o1, o2);** returns set of objects greater than o1 and less than o2
* **comparator();** returns comparator which returns underlying sorting. default null
* **TreeSet**
* Underlying data structure is a balanced tree
* Duplicates are not allowed.
* Insertion order not preserved.
* All elements will be inserted according to some sorting order.
* Heterogenous objects are not allowed.
* null insertion is possible but only once..
* **TreeSet Constructors**
* **TreeSet t = new TreeSet();** creates empty TreeSet objects and elements will inserted according to default natural sorting order.
* **TreeSet t = new TreeSet(Comparator c);** creates empty TreeSet objects and elements will inserted according to customized sorting order described by comparator.
* **TreeSet t = new TreeSet(Collection c);**
* **TreeSet t = new TreeSet(SortedSet s);**
* **Null acceptance in TreeSet**
* null can be inserted only once that too in the first insertion.
* Once null is inserted, other element cant be inserted because elements cant be compared
* **Miscellaneous observations TreeSet**
* If the TreeSet under consideration is using default natural sorting order, then the objects that can be added to it must be homogenous and comparable. StringBuffer example. Otherwise we get ClassCastException.
* Object is said to be comparable if it implements Comparable interface.
* **Comparable interface**
* Present in java.lang package
* Comparable interface contains only method: compareTo(Object o1)



* Either of the 2 objects cant be null
* When trying to add elements to TreeSet and depending on natural sorting order, JVM by default used compareTo() method.
* While using compareTo(), the element to be added to the TreeSet will be obj1 and the element already in TreeSet will be obj2
* Comparable meant for default sorting order.
* Comparator interface