**Collections**

Collection are growable in nature. i.e. based on our requirement we can increase or decrease the size.

Collection can hold both homogenous and heterogenous objects.

Every collection class is implemented based on some standard data structure. Hence for every method ready made method support is available being a programmer we are responsible to use those methods and we are not responsible to implement those methods.

Difference between arrays and collections:

|  |  |
| --- | --- |
| **Array** | **Collections** |
| 1. Arrays are fixed in size i.e. once we create an array we can’t increase or decrease the size based on our requirement. 2. With respect to memory arrays are not recommended to use. 3. With respect to performance arrays are recommended to use. 4. Arrays can hold only homogenous data type element. 5. There is no undelaying data structure for arrays and hence readymade method support is available.   For every requirement we have to write the code explicitly which increases complexity of programming.   1. Arrays can hold both primitives and objects. | 1. Collection are growable in nature i.e. based on our requirement we can increase or decrease the size. 2. With respect to memory collections are recommended to use. 3. With respect to performance collections are not recommended to use. 4. Collections can hold both homogenous and heterogenous element. 5. Every collection class is implemented based on some standard data structure and hence for every requirement readymade method support is available being a programmer we can use these methods directly and we are not responsible to implement those methods. 6. Collection can hold only object type not primitive type |

**Collection:**

If we want to represent a group of individual objects as a single entity then we should go for collection.

**Collection Framework:**

It contains several classes and interfaces which can be used to represent a group of individual objects as a single entity.

9 Key interfaces of collection framework:

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

**Collection(I):**

-> If we want to represent a group of individual object 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.

Difference between Collection and Collections

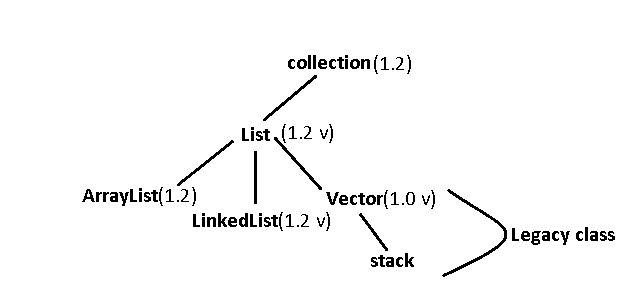
Collection is an interface, if we want to represent a group of individual objects as a single entity then we should go for it.

Collections is an utility class present in java.util package to define several utility methods for several collection objects(sorting, searching etc.).

**List(I):**

-> It is the child interface of collection.

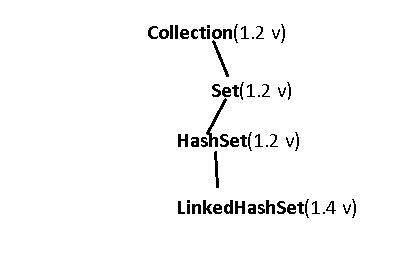
-> If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order must be preserved then we should go for list.



**Set (I):**

-> It is child interface of Collection.

-> If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order is not preserved then we should go for set.



**SortedSet (I):**

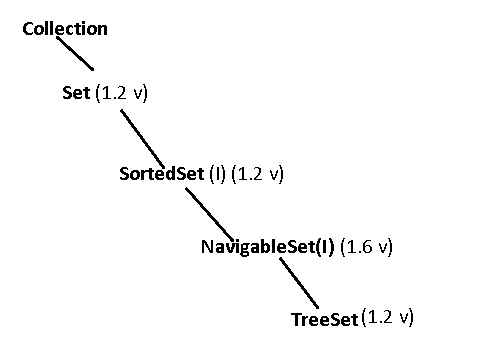
It is the child interface of set.

If we want to represent a group of individual object as a single entity where duplicates are not allowed and all objects should be inserted according to some sorting order.

**NavigableSet (I):**

It is the child interface of sorted set.

It contains several methods for navigation purposes.



Differences between list and set

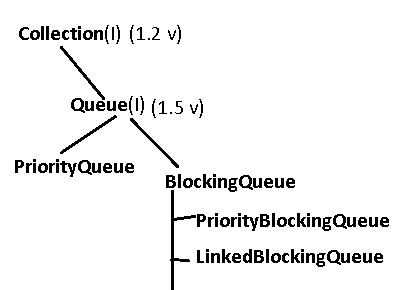
|  |  |
| --- | --- |
| **List** | **Set** |
| 1. Duplicates are allowed. 2. Insertion order preserved. | 1. Duplicates are not allowed. 2. Insertion order not preserved. |

**Queue:**

It is the child interface of collection.

If we want to represent a group of individual object prior to processing then we should go for queue.

Usually queue follows First In First Out order but based on our requirement we can implement our own priority queue also.



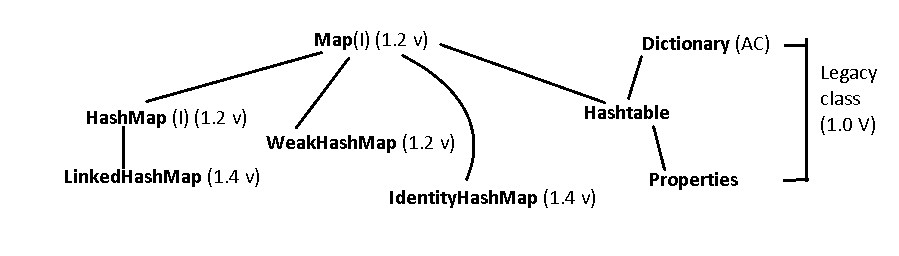
All the above interfaces (Collection, List, Set, SortedSet, NavigableSet, Queue) meant for representing a group of individual objects. If we want to represent a group of objects as key-value pairs then we should go for Map.

**Map**(I)**:**

-> Map is not child interface of collection.

-> If we want to represent a group of objects as key value pairs then we have to go for Map.

-> Both key and values are objects only. Duplicate keys are not allowed, but values can be duplicated.



**SortedMap(I):**

-> It is the child interface of Map.

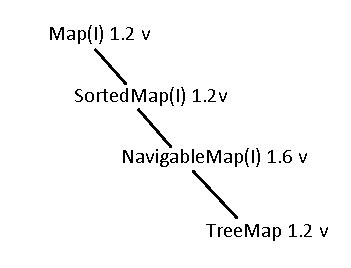
-> If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap.

-> Here sorting is done based on keys not based on values.

**NavigableMap**(I):

-> It is the child interface of SortedMap.

-> It defines several methods for navigation purposes.



**Note:** The following are legacy class present in collection framework

1. Enumeration (I)
2. Dictionary (AC)
3. Vector (C)
4. Stack (C)
5. Hashtable (C)
6. Properties (C)

Collection :

If we want to represent a group of individual objects as a single entity then we should for collection.

Collection interface defines the most common methods which are applicable for any collection objects.

1. boolean add(Object o)
2. boolean addAll(Collection c)
3. boolean remove(Object o)
4. boolean removeAll(Collection c)
5. boolean retainAll(Collection c) //to remove all except c
6. void clear()
7. boolean contains(Object o)
8. boolean containsAll(Collection c)
9. boolean isEmpty()
10. int size()
11. Object[ ] toArray( );
12. Iterator iterator( )

There is no concrete class which implements collection interface directly.

List(Interface):

List 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.

We can preserve insertion via index and we can differentiate duplicate objects via index hence index will play very important role in list.

List interface defines the following specific method:

1. void add(int index, Object o)
2. boolean addAll(int index, Collection c)
3. Object get(int index)
4. Object remove(int index)
5. Object set(int index, Object new) // To replace the element present at specified index with provided Object and returns old object
6. int indexOf(Object o) // returns index of first occurrence of ‘o’
7. int lastIndexOf(Object o) // returns index of last occurrence of ‘o’
8. ListIterator listIterator();

**ArrayLIst:**

-> The underlaying data structure is resizable array or growable array.

-> Duplicate objects are allowed.

-> Insertion order is preserved.

-> Heterogenous objects are allowed(Except TreeSet and TreeMap everywhere heterogenous objects are allowed).

-> null insertion is possible.

**Constructors:**

1. ArrayList l = new ArrayList();

Creates an empty arraylist objects with default initial capacity 10.

Once ArrayList reaches its max capacity then a new Arraylist will be created with

**New Capacity = (Current capacity \* 3/2) + 1**

1. ArrayList l = new ArrayList(int initialCapacity);

It creates an empty arraylist with specified initial capacity.

1. Arraylist l = new ArrayList(Collection c);

Creates an equivalent Arraylist object for the collection.

Usually we can use collection to hold and transfer objects from one location to another location to provide support for this requirement every collection class by default implements serializable and cloneable interface.

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 contains any method.

It is a marker interface where required ability will be provided by the JVM.

Ex:



O/P:

true

true

true

false

Arraylist is the best choice if our frequent operation is retrival (as it implements RandomAccess ).

ArrayList is the worst choice if our frequent operation is insertion and deletion at middle.

Difference between ArrayList and vector.

|  |  |
| --- | --- |
| ArrayList | Vector |
| 1. Every method present in the arraylist is non-synchronized. 2. At a time multiple threads are allowed to object on ArrayList objects and hence it is not thread safe. 3. Relatively performance are high because threads are not required to wait to operate on arraylist objects. 4. Introduced in 1.2 version and it is non-legacy. | 1. Every method present in the vector is synchronized. 2. At a time only one object are allowed to operate on Vector object and hence it is thread safe. 3. Relatively performance is low because threads are required to wait to operate on vector object. 4. Introduced in 1.0 version and it is legacy. |

Q. How to get synchronized version of ArrayList object?

By default ArrayList is non-synchronized but we can get synchronized version of arraylist objects by using **synchronizedList()**

Syntax – public static List synchronizedList(List l)

Ex: ArrayList l = new ArrayList();

List l1 = Collections.synchronizedList(l);

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 underlaying data structure is **Doubly Linked List**.

-> Insertion order is preserved.

-> Duplicate objects are allowed.

-> Heterogenous objects are allowed.

-> null insertion is possible.

-> LinkedList implements Serializable and Cloneable interface but not RandomAccess.

-> LinkedList is the best choice If our frequent operation is insertion or deletion in the middle.

-> LinkedList is the worst choice if our frequent operators is retrival.

**Constructors:**

1. LinkedList l = new LinkedList();

It creates an empty LinkedList object.

1. LinkedList l = new LinkedList(Collection c);

Creates an equivalent linked list object for the given collection.

LinkedList class specific methods:

Usually we can use LinkedList to develop stacks and queue. To provide support for support for this requirement LinkedList class defines the following specific methods.

1. void addFirst(Object o)
2. void addLast(Object o)
3. Object getFirst()
4. Object getLast()
5. Object removeFirst()
6. Object removeLast()

Ex: import java.util.LinkedList;  
public class Main {  
 public static void main(String[] args) {  
 LinkedList l = new LinkedList();  
 l.add("durga");  
 l.add(30);  
 l.add(null);  
 l.add("durga");  
 l.set(0,"software");  
 l.add(0,"Venky");  
 l.removeLast();  
 l.addFirst("ccc");  
 System.*out*.println(l);  
 }  
}

O/P:

[ccc, Venky, software, 30, null]

**Vector:**

->The underlaying data structure is resizable array or growable array.

-> Insertion order is preserved.

-> Duplicates are allowed.

-> Heterogenous objects are allowed.

-> Null insertion is possible.

-> It implements Serializable, Cloneable and RandomAccess interfaces.

-> Every method present in the vector is synchronized and hence vector object is thread safe.

**Constructors:**

1. Vector v = new Vector();

It creates an empty vector object with default initial capacity 10.

Once it reaches the maximum capacity then a new vector object will be created with new capacity = current capacity \* 2

1. Vector v = new Vector(int initialCapacity);

Creates an empty vector object with specified initial capacity.

1. Vector v = new Vector(int initialCapacity, int incrementalCapacity);
2. Vector v = new Vector(Collection c);

Creates an equivalent vector object for the given collection. This constructor meant for interconversion between collection object.

Vector Specific Method:

1. addElement(Object o)
2. removeElement(Object o)
3. removeElement(int index)
4. removeAllElements()
5. Object elementAt(int index)
6. Object firstElement( )
7. Object lastElement( )
8. Enumeration elements()
9. int size()
10. int capacity

Ex: import java.util.Vector;  
public class Main {  
 public static void main(String[] args) {  
 Vector v = new Vector();  
 System.*out*.println(v.capacity());  
 for (int i=0;i<=10;i++){  
 v.addElement(i);  
 }  
 System.*out*.println(v.capacity());  
 v.add("durga");  
 System.*out*.println(v.capacity());  
 System.*out*.println(v);  
 }  
}

O/P:

10

20

20

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, durga]

**Stack:**

It is the child class of vector.

It is a specially designed class for last in first out order.

**Constructors:**  Stack s = new Stack();

Methods:

1. Object push(Object o)

to insert an object into the stack and returns that object .

1. Object pop()

to remove and return top of the stack.

1. Object peek()

to return top of the stack without removal.

1. boolean empty()

returns true if the stack is empty.

1. int search(Object o)

If object is found then it returns offset(position of element from top in the stack) otherwise it will return -1.

Ex:

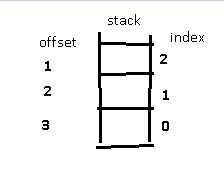
import java.util.Stack;  
public class Main {  
 public static void main(String[] args) {  
 Stack s = new Stack();  
 s.push("A");  
 s.push("B");  
 s.push("C");  
 System.*out*.println(s);  
 System.*out*.println(s.search("A"));  
 System.*out*.println(s.search("Z"));  
 }  
}

O/P:

[A, B, C]

3

-1



**The 3 cursor of java:**

If want to get objects one by one from the collection then we should go for cursor.

There are 3 types of cursor available in java.

1. Enumeration
2. Iterator
3. ListIterator

**Enumeration:**

We can use Enumeration to get object one by one from legacy collection object.

We can create Enumeration objects by using elements() method of vector class.

**public Enumeration elements();**

Ex:

Enumeration e = v.elements();

**Methods:**

1. public boolean hasMoreElement();
2. public Object nextElement();

Ex:

package com.company;  
import java.util.Enumeration;  
import java.util.Vector;  
public class Main {  
 public static void main(String[] args) {  
 Vector<Integer> v = new Vector<Integer>();  
 for (int i=1;i<11;i++){  
 v.addElement(i);  
 }  
 System.*out*.println(v);  
 Enumeration e = v.elements();  
 while(e.hasMoreElements()){  
 int i =(Integer)e.nextElement();  
 if(i%2 == 0)  
 System.*out*.println(i);  
 }  
 }  
}

Limitations of Enumeration:

* we can apply enumeration concept only for legacy classes and it is not an universal cursor.
* By using enumeration we can get only read access and we can’t perform remove operation.

To overcome these problems we should go for Iterator.

**Iterator:**

We can apply iterator object for any collection object and hence it is universal cursor.

By using Iterator we can perform both read and remove operations.

We can iterate object by using iterator() method of collection interface.

**public Iterator iterator()**

Ex:

Iterator itr = c.iterator(); // c – Any collection object

Methods:

1. public boolean hasNext()
2. public Object next()
3. public void remove()

Ex:

package com.company;  
import java.util.Iterator;  
import java.util.Vector;  
public class Main {  
 public static void main(String[] args) {  
 Vector<Integer> v = new Vector<Integer>();  
 for (int i=1;i<11;i++){  
 v.addElement(i);  
 }  
 System.*out*.println(v);  
 Iterator itr = v.iterator();  
 while(itr.hasNext()) {  
 int i = (Integer) itr.next();  
 if (i % 2 == 0)  
 System.*out*.println(i);  
 else {  
 itr.remove();  
 }  
 }  
 System.*out*.println(v);  
 }  
}

Limitations of Iterator and Enumeration:

* These two cursor are single directional (Forward) cursor but not bi-directional cursors.
* By using iterator we can perform read and remove operation and we can’t perform replacement and addition of new objects.

To overcome above limitations we should go for Listiterator.

**ListIterator:**

By using ListIterator we can move either to the forward direction or to the backward direction and hence it is bidirectional cursors.

By using ListIterator we can perform replacement and addition of new object in addition to read and remove operations.

We can create ListIterator by using listIterator() of List interface.

**public ListIterator listIterator( )**

Ex:

ListIterator ltr = l.listIterator(); //l is Any List Object.

Methods:

ListIterator is the child interface of Iterator and hence all methods present in Iterator by default available to the ListIterator.

Forward Movement ->

1. public boolean hasNext()
2. public Object next()
3. public int nextIndex()

Backward Movement ->

1. public boolean hasPrevious()
2. public Object previous()
3. public int previousIndex()

Extra Modification operations ->

1. public void remove()
2. public void add(Object o)
3. public void set(Object o)

Ex:

package com.company;  
import java.util.LinkedList;  
import java.util.ListIterator;  
public class Main {  
 public static void main(String[] args) {  
 LinkedList l = new LinkedList();  
 l.add("balakrishna");  
 l.add("venki");  
 l.add("chiru");  
 l.add("nag");  
 System.*out*.println(l);  
 ListIterator ltr = l.listIterator();  
 while(ltr.hasNext()){  
 String s = (String)ltr.next();  
 if(s.equals("venki")){  
 ltr.remove();  
 }else if(s.equals("nag")){  
 ltr.add("chaitu");  
 }else if (s.equals("chiru")){  
 ltr.set("charan");  
 }  
 }  
 System.*out*.println(l);  
 }  
}

The most power full cursor is ListIterator but its limitation is it is applicable only for List objects.

**Internal Implementation of Cursors:**

package com.company;  
import java.util.\*;  
public class Main {  
 public static void main(String[] args) {  
 Vector v = new Vector();  
 Enumeration e = v.elements();  
 Iterator i = v.iterator();  
 ListIterator ltr = v.listIterator();  
 System.*out*.println(e.getClass().getName());  
 System.*out*.println(i.getClass().getName());  
 System.*out*.println(ltr.getClass().getName());  
 }  
}

O/P:

java.util.Vector$1

java.util.Vector$Itr

java.util.Vector$ListItr