## Collections

## Array:

Array is an indexed collection of fixed number of homogenous data elelments.

Limitations:

1. Fixed in size
2. Homogenous data type elements
3. Arrays concept is not based on standard data structure(no underlying data structure), thus we cannot expect readymade method support.
   Eg: sorting/searching

## Collections frame work:

Advantages:

1. Growable in nature
2. Homogenous and Heterogeneous both.
3. Every collection class is based on standard Data structure, hence readymade method support is available.

Difference between Arrays and collection

| Arrays | Collections |
|---|---|
| Fixed | Growable |
| Wrt to memory Arrays are not recommended to use. | Wrt to memory collections are recommended to use. |
| Wrt to performance Arrays are recommended | Wrt to performance collections are not recommended. |
| Arrays can hold only homogenous data | Collections can hold homogenous and heterogeneous both. |
| Arrays are not created based on standard Data structure, hence readymade method is not available. | Collections are based on standard data structure, hence readymade method is available. |
| Array can hold primitive and objects | Collections can hol only object but not primitives. |

## Collection:

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

Collection Framework:

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

9key interfaces of collection framework:

### 1. Collection

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

- Collection interface defines most common methods which are applicable for any collection object.
- In general collection is considered as root framework of collection Framework.

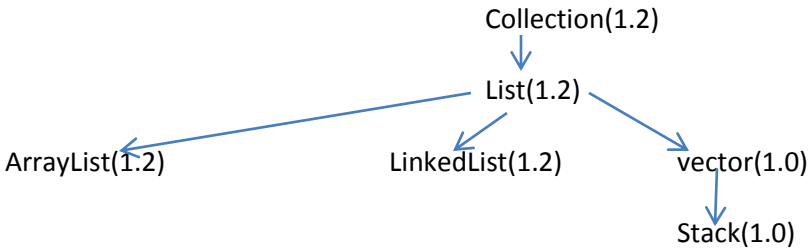
Note: there is no concrete class which implement collection interface directly

Different between Collection and Collections

| Collection | Collections |
|---|---|
| It is a interface used to represent a group of individual objects as a single entity. | It is a utility class which defines several utility method collectionObject.(like sorting searching) |

## 2. List:

- 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 is preserved then we should go for List.




*vector and stack are called legacy classes

## 3. Set:

- It is the child interface of collection.
- If we want to represent a group of object a single entity wher duplicates are not allowed and insertion order is not preserved then we should go for Set.
  Collection(1.2)→Set(1.2)→HashSet(1.2)→LinkedHashSet(1.4)

Difference between List and Set

| **List** | **Set** |
|---|---|
| Duplicates are allowed | Duplicates are not allowed |
| Insertion order is preserved | Insertion order is not preserved |

## 4. SortedSet:

- 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 but all objects should be inserted according to some sorting order then we should go for sorted set.

Collection→Set→SortedSet(1.2)

5. **NavigableSet**
    - It is the child interface of sortedSet.
    - It defines several methods for navigation purpose.
    - Its implementation class is treeSet.

    Collection→Set→SortedSet→Navigableset(1.6)→TreeSet(1.2)

6. **Queue**
    - It is the child interface of collection
    - If we want to represent a group of individual objects prior to processing then we should go for Queue.

    Eg: Before sending email you need to store all email Ids somewhere and should be able to retrieve in the same order(FIFO) for this requirement Queue concept is best choice.

Collection→Queue→PriorityQueue
→BlockingQueue →LinkedBlockingQueue and PriorityBlockingQueue
Entire Queue concept came in ver 1.5

Note: All above interfaces are meant for representing group of individual objects. If we want to represent a groups of objects as key-value pair we should go for Map interface.

7. **Map**
    - Map is not the child interface of collection.
    - If we want to represent a group of objects as a key –value pair then we should go for Map.
      Both Key and value are objects, however duplicate keys are not allowed but duplicate values are allowed.

      Implementation classes:
      Map
        - HashMap(1.2)
            - LinkedHashMap(1.4)
        - WeakHashMap(1.2)
        - IdentityHasMap(1.4)
        - Dictionary(1.0)→Hashtable(1.0)
            - Properties(1.0)

Note:Dictionary, Hashtable,properties are leagacy classes.

8. **SortedMap**
    - It is the child class of Map.
    - If we want to represent a group of objects as a key-value pair according to some sorting order of keys then we should go for SortedMap.
      Map→Sortedmap(1.2)
9. **NavigableMap**
    - It is the child interface of sorted Map
    - It defines several utility method for navigation purpose.

NavigableMap(1.6)→Treemap(1.2)

Sorting:

Comparable vs comparator:

| Comparable | comparator |
|---|---|
| Natural sorting | User defined Sorting |

Cursors:

Enumerations
Iterator
ListIterator

Utilityclasses

Collections
Arrays

**Collection Interface:**

1. If we want to represent a group of individual objects as a single entity, then we should go for collection.
2. In general collection interface is considered as root interface of collections framework.
3. Collection interface defines the most common methods which applicable for any collection object.

   Important methods of collection interface:

   Boolean add(Object o);
   Boolean addAll(Collection c)
   Boolean remove(Object o)
   Boolean removeAll(Collection c)
   Boolean retainAll(Collection c)
   Void clear()
   Boolean contains(Object o)
   Boolean containsAll(Collection c)
   Boolean isEmpty()
   Int size()
   Object[] toaArray()
   Iterator iterator()

   Note: Collection interface does not contain any method to retrieve objects. There are no concrete class that implements collections.

**List Interface**

1. It is the child interface of Collection.
2. If we want to represent a group of objects as a single entity, where duplicates are allowed and Insertion order must be preserved we should go for List.
3. Duplicates can be identified by using index.
4. Insertion order is preserved using index, thus index play a very important role in in List.

   Important methods of List interface:

   Void add(int index,Object o)
   Boolean addAll(int index,Collection c)
   Object get(int index)
   Object remove(int index)
   Object set(int index,Object o)
   Int indexOf(Object o)
   Int lastIndexOf(Object o)
   ListIterator listIterator();

- **ArrayList:**
  1. The underlined datastructure is Resizable array or Growable array
  2. Duplicates are allowed
  3. Insertion order is preserved
  4. Heterogeneous objects are allowed(except TreeSet and TreeMap)
  5. Null insertion is possible.

Constructors of Arraylist:

1. ArrayList al =new ArrayList();
   Creates an empty ArrayList object with ***default initial capacity 10***.
   Once array list reaches max capacity new ArrayList will be created with new capacity = (current capacity*3/2)+1.
2. ArrayList al = new ArrayList(int initialCapacity)
3. ArrayList al = new ArrayList(Collection c); // interconversion for eg Linked to arraylist or vector to arraylist.
4. ArrayList and Vector classes implement RandomAcess interface, so that we can access any Random element with same speed.
5. If frequent operation is retrieval operation then ArrayList is best choice.
6. If frequent operation is insertion or deletion in middle ArrayList is worst choice

   Note: usually we use collections to hold and transfer objects from one place to another, to meet this requirement every Collection already implements serializable and cloneable interfaces.

   RandomAcess: Present in Java.Util , it does not contain any method it is a marker interface.

   Difference between ArrayList and Vector:

| ArrayList | Vector |
|---|---|
| Every method present is ArrayList is non –synchronized | Every method in Vector is synchronized |
| Not Thread safety | Thread safety |
| Relatively high performance | Relatively low performance |
| Introduced in ver 1.2, hence not legacy class | It is a legacy class |

   How to get synchronized version of ArrayList:
   Public static List synchronizedList(List l);
   Public static List synchronizedSet(Set s);
   Public static List synchronizedMap(Map m);

   ArrayList al =new ArrayList();
   ArrayList l=Collections.synchronizedList(al);

- **LinkedList:**
  1. The underlying data structure is double linked list.
  2. Insertion order is preserved.
  3. Duplicates are allowed.
  4. Null Insertion is possible.
  5. LinkedList implements serializable and cloneable but not RandomAccess.
  6. LinkedList is best choice if our frequent operation is insertion or deletion in middle.
  7. LinkedList is worst choice if our frequent operation is retrival operation.

Important methods of LinkedList:

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

Constructors:

1. LinkedList l1=new LinkedList(); // creates empty linkedlist
2. LinkedList l=new LinkedList(Collection c);

Difference between ArrayList and LinkedList.

| ArrayList | LinkedList |
|---|---|
| It is best choice if our frequent operation is retrival | It is best choice when our frequent operation is insertions or deletion in middle |
| Underlying data structure Is Resizable or Growable array | Underlying data structure is double LinkedList |
| ArrayList implements RandomAccess Interface | LinkedList doesn't implement Random Acces Interface |

- **Vector**

1. The underlying data structure is Resizable or Growable array
2. Duplicates are allowed.
3. Insertion order is preserved
4. "null" insertion is possible
5. Heterogeneous objects are allowed
6. Vector class implements serializable and cloneable and RandomAccess.
7. Most of the method are synchronized, hence thread safe.
8. Best choice when frequent operation is retrival.

Important methods of Vector:
For adding data,
Add(Object o)
Add(int index,object o)
addElement(Object o)

for removing
remove(Object o)
removeElement(Object o)
remove(int index)
clear()
removeAllElement()

to retrieve
Object get(int index)
Object getElement(int index)

Object getFirstElement()
Object getLastElement()

Other methods,
Int size()
Int capacity()
Enumeration element ()

Constructors f vector

1. Vector v =new Vector()
   Creates empty vector with **default initial capacity of 10**.The capacity increases as below,
   New capacity = 2 * currentcapacity
2. Vector v =new Vector(int initialcapacity)
3. Vector v=new Vector(int initialCapacity,int incremenatlcapacity)
4. Vector v = new Vector(Collection c);

- **Stack:**
  1. It is the child class of Vector.
  2. Specially designed for Last In First Out

Constructor:
Stack s=new Stac();

Important methods of Stack:

1. Push(object 0) //add
2. Pop() //remove and return top of stack
3. Peek() //return top of stack
4. Int search(object o) //returns offset
5. Empty() //to check if stack is empty

## Cursors of Java

If we want to retrieve objects from collection one by one then we should go for cursors.
Three cursors in java:

- Enumeration
- Iterator
- ListIterator

Enumeration:
We can use Enumeration to get objects from legacy collection.
Enumeration e= v.elements(); //v is any vector

e.hasMoreElements()
e.nextElement()

Iterator:
We can apply iterator concept for any collection object hence it is universal cursor.
By using iterator we can perform both read and remove operation.

Iterator itr=c.iterator(); //c is any collection

Itr.hasnext()
Itr.next()
Itr.remove()

Limitations

1. Only forward direction cursors
2. We cannot add or Replace.

ListIterator:

1. ListIterator is bi-directional
2. We can perform, read, remove, replacement and addition of objects.

How to get ListIterator object
ListIterator ltr= l.listIterator(); //l is any list.

Methods:
hasNext();
next();
nextIndex();

hasPrevious();
previous();
previousIndex();

remove();
set(object new);
add(Object new);

| Property | Enumeration | Iterator | ListIerator |
|---|---|---|---|
| Applicable for | Legacy | Any | Only list |
| Movement | Forward only | Forward only | Bi-directional |
| Accessibility | Read only | Both read and remove | Read, remove,replace,add |
| How to get it | Elements() | Iterator() | listIterator() |
| methods | HasMoreElement()<br>Next() | hasNext();<br>next();<br>remove(); | 9 methods |
| Is it legacy | Yes | No | no |

Implementation classes of three cursors:
There are internal inner class wich implements above interface.

## Set Interface:

1. It is the child interface of collection
2. If we want to represent a group on individual objects as a single entity where duplicates are not allowed and insertion order is not preserved then we should go for set.
3. Set interface doesn't contain any new method, so we have to use only collection interface methods.

HashSet:

1. The underlying data structure is hashtable.
2. Duplicates are not allowed. Add () method returns false.
3. Insertion order is not preserved but is based on hash-code of objects.
4. Heterogenous objects are allowed
5. 'null' insertion is possible.
6. Implements serializable() and cloneable()
7. Best choice for search operations.

Constructors:

1. HashSet h=new HashSet();
   -creates an empty hashset with default initial capacity of 16 and default fill ration:0.75
2. HashSet h=new HashSet(int initialCapacity);
3. Hashset h=new HashSet(int initialcapacity,float incrementalcapacity)
4. HashSet h = new HashSet(Collection c);

LinkedHashSet:

1. Child class of hashSet.
2. Introduced in version 1.4

| HashSet | LinkedHashset |
|---|---|
| Based on Hashtable | Based on linkedList+hashtable |
| Insertion order is not preserved | Insertion order is preserved |
| 1.2 | 1.4 |

SortedSet:

1. It is the child of set interface.
2. If we want to represent a group of individual objects according to some sorting and duplicates are not allowed we should go for sorted set.
3. Methods:
   First()→returns first element of the set.
   Last()→returns last element of the set.
   headset(object o)→returns sortedset whose elements are <object
   tailSet(Object o)→ returns sortedset whose elements are >=object
   subset(object o1,object o2)→>=o1 and <o2

   comparator()→returns compartor object that describes the underlying sorting technique, if its natural sorting we get null.

TreeSet:

1. The underlying data structure for tree set is balanced tree.
2. Duplicates are not allowed.
3. Insertion order is not preserved.
4. Heterogenous objects are not allowed(RE: classCastException)
5. Null is allowed but only once

Constructors:
TreeSet t=new TreeSet();

- Creates empty treeset

TreeSet t= new TreeSet(comparator c);

- Customized sorting.

TreeSet t= new TreeSet(collection c);
TreeSet t=new Treeset(sortedset s);

For empty tree set null insertion is possible as a the first element but if we are trying to insert after that we get null pointer exception.

Note:
If we are depending on default natural sorting order then objects should be homogenous and comparable, else we get classCastException.
An object is comparable if and only if corresponding class implements comparable interface.
String and all wrapper classes implements comparable.

Comparable Inerface:
It is present in java.lang packageit contains only one method CompareTo(object obj)

Example:
Obj1.compareTo(obj2)
→returns -ve if obj1<obj2
→returns +ve if obj1>obj2
→returns 0 if obj1 and obj2 are equal

Comparator interface:
We can use comparator interface to define our own sorting.
Compartor interface is present in java.util package.
It defines two methods compare() and equal()

1. Public int compare(object obj1, object obj2)//same as compareTo();
2. Public boolean equals()

Public int compare(obj I1,obj i2)
//return i1.compareTo(i2)→ ascending order
//return –i1.compareTo(i2)→descending
//return i2.compareTo(i1)→descending
//return –i2.compareTo(i1)→ascending
//return +1 insertion order is preserved.

//return -1 insertion order is reversed.
//retrun 0 only first element is inserted.

Comparable vs comparator:

| Comparable | Comparator |
|---|---|
| D.N.S.O | Customized sorting order |
| Java.lang | Java.util |
| compareTo() | Compare() and equals() |
| All wrapper and sring class implements comparable | No pre-defined classes, |

## Map:

1. Map is **not a child** interface of collection.
2. If we want to represent a group of objects as key value pairs, then we should go for Map.
3. Both key and values are objects ony.
4. Duplicates keys are not allowed but values can be duplicated.
5. Each key value pair is called Entry. Hence Map is considered as a collection of Entry Objects.

### Methods:

1. Object put(object key, object value)→ to add one key value pair to the Map.
   If the key is already present then old value will be replaced with new value and returns old value.
2. Void putAll(map m)
3. Object get(Object key)→returns the value associated with specified key.
4. Object remove(Object key)→ removes entire entry
5. Boolean containskey(Object key)
6. Boolean containsValue(Object value)
7. Boolean isEmpty()
8. Int size()
9. Void clear()

Collection views of map:

10. Set keyset()
11. Collection values()
12. Set entrySet()

### Entry Interface:

A map is a group of key value pairs and each key value pair is called an entry, hence map is considered as a collection Entry objects. Without existing Map object there is no chance of existing entry object. Hence Entry interface is defined inside the Map interface.

```
Interface Map{
    Interface Entry{
            Object getKey()
            Object getValue()
            Object setValue(Object new)
```

```
    }
}
```

## HashMap:

1. The underlying datastructure is Hashtable.
2. Insertion order is not preserved and it is based on Hashcode of keys.
3. Duplicates keys are not allowed.
4. Duplicate values are allowed.
5. Heterogenous objects are allowed for both key and value.
6. Null is allowed for key but only once, no restriction for values.
7. Implements serializable and cloneable.
8. Best used for searching.

Constructors:

1. HashMap m=new HashMap()→ create an empty hashmap oject with default initial capacity 16 and default fill ratio 0.75
2. Hashmap m=new HashMap(int initialcapacity)
3. Hashmap m=new Hashmap(int initialcapacity, float fillratio)
4. Hashmap m =new Hashmap(map m)

Difference between Hashmap and Hashtable

| Hashmap | Hashtable |
|---|---|
| Not synchronized | synchronized |
| Not Thread safe | Thread safe |
| Relative performance is high | Relative performance is low |
| null is allowed | Null concept is not allowed |
| Its not legacy class | It is a legacy class |

By default hashmap is non-synchronized but we can get the synchronized version of hashmap by using synchronized map method of collections class

```
Hashmap m=new Hashmap()
Map m1=collections.synchronizedmap(m);
```

## LinkedHashmap:

1. The underlying structure is Hashtable+linkedlist
2. Insertion order is preserved
3. Introduced in 1.4 ver
4. Linked hashset and linkedHashmap are commonly used for developing cache based applications

## IdentityHashmap:

Normal HashMap uses .equals() method whereas IdentityHashmap uses “==” which is meant for refrence comparision

## WeakHashmap:

It is exactly same as hashmap except the following difference.
In case of hashmap eventhough object doesn't have any reference it is not eligible for GC if it is associated with hashMap.
In case of weakHashmap, if object doesn't contain any references it is eligible for GC eventhough object associated with weakHashmap

## Sortedmap:

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 sorted map.

Methods of sortedmap:

1. firstKey()
2. lastKey()
3. headMap(object key)
4. tailMap(Object key)
5. subMap(Object k1,object k2)
6. comparator()

## TreeMap:

1. Underlying datastructure is RED-BLACK tree.
2. Insertion order is not preserved and it is based on some sorting order of keys
3. Duplicate keys are not allowed but Duplicate values are allowed.
4. If we are depending on default natural sorting order then keys should be homogenous and comparable otherwise we will get classCastException.
5. If we are defining our own sorting by comparator then keys need not be homogenous and comparable, we can take heterogeneous non-comparable objects also.
6. There are no restrictions for values.
7. Null can be inserted only as a first entry in empty TreeMap
   From ver 1.7 null is not accepted for treemap.

   Consructors:
   TreeMap m=new TreeMap();
   TreeMap m=new TreeMap(comparator c)
   TreeMap m=new TreeMap(Map m1);
   TreeMap m=new TreeMap(SortedMap sm)

## Hashtable:

1. The underlying data structure for hashtable is hashtable.
2. Insertion order is not preserved and it is based on hashcode of keys.
3. Duplicate keys are not allowed and values can be duplicated.
4. Heterogeneous objects are allowed for both keys and values.
5. Null is not allowed for either.
6. It implements serializable and cloneable interfaces but not RandomAcess.
7. Every method present in hashtable is synchronized and hence Hashtable object is thread safe.

8. Hashtable is best choice if our frequent operation is search operation.

Constructors:
Hashtable h=new Hashtable() → creates an empty Hashtable with default initialcapacity 11 and fill ratio 0.75
Hashtable h =new Hashtable(int initialCapacity)
Hashtable h=new Hashtable(int initialcapacity,float fillratio)
Hashtable h=new Hashtable(map m)

Properties:

## Queue:

It is the child interface of collection. If we want to represent a group of individual objects prior to processing then we should go for Queue for eg: before sending sms message all mobile numbers we have to store in some data structure, in the same order in which we added mobile numbers , message should be delivered. For this his FIFO requirement queue is the best choice.

Usually queue follows FIFO but based on our requirement we can implement our own priority order also(priorityqueue)
From 1.5 ver onwards LinkedList class also implements queue interface.
LikedListbased implementation of queue always follows FIFO

Methods:

1. Offer(Object o)
2. Poll()→remove and return head of the queue
3. Remove()
4. Peek()→returns head element
5. Element()

## PriorityQueue:

1. If we want to represent a group of individual objects prior to processing according to some priority then we should go for priority queue.
2. The priority can be either D.N.S.O or customized sorting order defined by comparator
3. Insertion order is not preserved and it is based on some priority.
4. Duplicate objects are not allowed
5. If we are depending on D.N.S.O compulsory objects should be Homogeneous and comparable else we get runtime exception saying classCastException.
6. If we are defining our own sorting by comparator then objects need not be homogeneous and comparable.
7. Null is not allowed.

Constructor:
priorityQueue q=new PriorityQueue(); →empty queue with default capacity 11 and D.N.S.O
priorityQueue q=new priorityQueue(int initialcapacity);
priorityQueue q=new priorityQueue(int initialCapacity, comparator c);
priorityQueue q=new priorityQueue(sortedset s);

priorityQueue q=new priorityQueue(Collection c)

## 1.6ver enhancement in collection framework

As a part of 1.6 ver following two concepts introduced in collection framework
NavigableSet , NavigableMap

NavigableSet:
It is the child interface of sorted set and it defines several methods for Navigation purposes

Methods:
Floor(e)→returns highest element which is <= e
Lower(e)→ retruns highest element which is <e
Ceiling(e)→lowest element which is >=e
Higher(e )→ lowest element which is >e
Pollfirst()→ remove and return first element
pollLast()→remove and return last element
descendingSet ()→returns set in reverse order

NavigableMap:
NavigableMap is the child interface of sortedMap, it defines several methods for navigation purposes.

floorKey(e)
lowerKey(e )
ceilingKey(e )
higherKey(e )
pollFirstEntry()
pollLastEntry()
descendingMap()

Collections class search:
Pubic static int binarySearch(list l, object target);→D.N.S.O
Public static int binarySearch(List l,Object target,Comparator c)→customized sorting

Note: for the list of n elements, in the case of binarysearch method

1. Successful search result range→0 to n-1
2. UnScuccessful search result range→-(n+1) to -1
3. Total result range→ -(n+1) to (n+1)