Collections

- Collections are grow able in nature. Based on our requirement we can increase or decrease the size.
- Collections can hold both homogeneous and heterogeneous element.
- Every collection class implemented based on some standard data structure. Hence for every requirement readymade method support is available.
- Being a programmer we are responsible to use methods and we are not responsible to implement those methods
- Collections can hold only object data types but not primitives.
- If we want represent group of individual object as single entity then we should go for collection.

Collection Framework

It contains several classes and interfaces to represent group of individual object as single entity.

9 Key Interfaces

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

Collection (I)

If we want represent a group of individual object as a single entity then we should go for collection. Collection defines most common methods which are applicable for any collection object. In general collection interface is considered as root interface of collection framework.

Collections (C)

Collections is an utility class present in java.util package. To define several utility methods for collection object like sorting, searching, etc.

List(I)

It is a child interface of collection. If we want represent a group of individual object as a single entity where duplicate are allowed and insertion order must be preserved then we should go for list.

Collection -> List -> ArrayList -> LinkedList -> Vector -> Stack

Set (I)

It is a child interface of collection. If we want represent a group of individual object as a single entity where duplicates are not allowed and insertion order not required then we should go for Set.

SortedSet (I)

It is a child interface of collection. If we want represent a group of individual object as a single entity where duplicates are not allowed and all objects should be inserted to some sorting order then we should go sorted set.

NavigableSet (I)

It is a child interface of collection. It contains several methods for navigation purposes.

Collection -> Set -> SortedSet -> NavigableSet -> TreeSet

Queue (I)

It is a child interface of collection. If we want represent a group of individual objects prior to processing then we should go for Queue. Usually Queue Follows FIFO order based on our requirement. We can implement our own priority order also.

Eg – mail Service

Collection -> Queue -> PriorityQueue -> BlockingingQueue -> …..
PriorityBlockingQueue
LinkedBlockingQueue

Map

Map is not child Interface of collection. If we want to represent a group of object as KEY, Value pair then we should go for Map. Both Key and value are Object only. Duplicate keys are not allowed. But values can be duplicated.

Dictionary (AC)
Map -> HashMap -> WeakHashMap -> IdentityHashMap -> Hashtable
LinkedHashMap
Properties

SortedMap

It is a child interface of collection. If we want represent a group of Key, Value pair according to some sorting order of key.

NavigableMap

It is a child interface of SortedMap and it defines several methods for Navigation Purposes.

Map -> SortedMap -> NavigableMap -> TreeMap

Collection (I)

Common methods

- 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[] toArray();
- Iterator iterator();

List (I)

Common methods

- 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 new)
- int indexOf(object o)
- int lastIndexOd(Object o)
- ListIterator listIterator();

ArrayList

ArrayList l = new ArrayList creates empty arraylist object with default initial capacity. Once arraylist reaches its max capacity then new array list object will be created.

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

ArrayList l = new ArrayList[int initial capacity] creates empty array object with specified initial capacity.

ArrayList l = new ArrayList(Collection c) creates an equivalent ArrayList object for given Collection

Note

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

RandomAccess

It is present in java.util package. It doesn't contain any methods hence it is a marker interface. Where required ability will be provided automatically by JVM.

ArrayList and vector class's implements RandomAccess interface so that any random element we can access with the same speed.

ArrayList is best choice if our frequent operation is retrieval operation. Because it is implement random access interface.

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

By default ArrayList is non-Synchronized. We can get synchronized version of ArrayList object by using SynchronizedList method of collection class.

Public static List synchronizedList(List L);

```
ArrayList al = new ArrayList();
List l1 = Collections.synchronizedList(l1);
```

Similarly we can get Synchronized version of Set and Map object by using following methods of Collection class.

Public static Set synchronizedSet(Set s)
Public static Map synchronizedMap(Map m)

LinkedList is best choice if our frequent operation is insertion and deletion in the middle.

LinkedList is worst choice if our frequent operation is retrieval operation. Because it will travel from 0^{th} index to search required element.

LinkedList Constructors:

```
LinkedList l = new LinkedList();
LinkedList l = new LinkedList(Collection c);
```

LinkedList class specific methods

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

Void addFirst(Object o);
Void addLast(Object o);
Object getFirst();
Object getLast();
Object removeFirst();
Object removeLast();

Vector

Vector v = new Vector(); creates an empty vector object with default capacity **10**. Once vector reaches max capacity the new vector object will be created.

new capacity = current capacity *2

Vector v = new Vector(initial capacity); creates an empty vector object with initial capacity.

Vector v = new Vector(initial capacity , int incremental capacity);

Vector v = new Vector(Collection c);

Vector Specific methods

int capacity();
Enumeration elements();

Stack

It is a child class of vector. It is a specially designed class for LIFO order. It is insertion preserved. Only in removal process LIFO order

Constructors

Stack s = new Stack();

Methods

Object push(object o); // To insert an object into the stack.
Object pop(Object o); // To remove and return top of the stack.
Object peek(); //To return top of the stack without removal.
Boolean empty(); //return true if stack is empty.
int search(Object o); //returns **offset** if the element is available otherwise returns -1.
offset starts from {1,2,3….}

The three cursors of java

1. **Enumeration**
2. **Iterator**
3. **ListIterator**

Enumeration

We can use Enumeration to get objects one by one from legacy collection object. We can create Enumeration object by using enumeration method of Vector class.

Vector v = new Vector();
Enumeration e = v.elements();

Methods

Public Enumeration elements();
Public Boolean hasMoreElements();
Public Object next Element();

Limitation of Enumeration

- We can apply Enumeration concept only for legacy classes and it is not a universal cursor.
- We can get only read access and we cant remove operation,
- To overcome above limitation we should go for Iterator.

Iterator

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

- Iterator I = c.iterator();

Methods

Public Iterator iterator();
public Boolean hasNext();
public Object next();
public void remove();

Limitation of Iterator

- By using enumeration and Iterator we can always move towards forward direction and we can’t move backward direction. These are single direction cursor. But not bi-direction cursor.
- By using Iterator we cannot perform replacement and addition of new Objec.
- To overcome above limitation we should go fir list Iterator.

ListIterator

By using listIterator we can move forward and backward direction hence it is bi-directional cursor. By using listIterator we can perform replacement and additional of new objects in addition to read and remove operations.

ListIterator LI = l.listIterator(); // l = any list object

Methods

Public ListIterator listIterator();
public boolean hasNext();
public Object next();
public int nextIndex()

public Boolean hasPrevious()
public Object previous()
public int previousIndex()

Public void remove();
public Object previous();
public void set(Object o);

Limitation of ListIterator

- It is applicable only for list objects.

Set

Collection -> Set -> HashSet -> SortedSet
LinkedHashSet NavigableSet
TreeSet

Note

Set interface doesn't contain any new method only we have to use collection interface method.

HashSet

The Underlined data Structure is hashTable. All objects are inserted based on hashCode. Implements Serializable and cloneable but not RandomAccess. It is best choice if our frequent operation is Search operation.

Note

In hashSet duplicates are not allowed if we are trying to insert duplicates then we won't get any compile time or runtime errors. And add method simply returns false.

HashSet Constructors

HashSet h = new HashSet(); creates an empty HashSet object with default initial capacity with 16 and default fill ratio is 0.75.

HashSet h = new HashSet(int initialCapacity); Creates an empty HashSet object with specified initial capacity and default filled ratio is 0.75;

HashSet h = new HashSet(int initialCapacity , float fillledRatio); Creates an empty HashSet object with specified initial capacity and specified filled ratio.

HashSet h = new HashSet(Collection c); this constructor meant for inter-conversion between collection object.

Fill Ratio / Load Factor

After filling how much ratio a new HashSet object will be created. This ratio is called filled ratio or load factor. Filled ratio 0.75 means after filling 75% ratio, a new HashSet object will be created.

LinkedHashSet

It is a child class of HashSet and it is exactly class as HashSet including constructor and methods except the following differences

- The underlying data Structure is combination of linkedList and HashTable.
- Insertion Order preserved.

Note

In general we can use linked HashSet to develop cache based application where duplicates are not allowed and insertion order is preserved.

SortedSet (I)

It is child interface of Set. If we want to represent a group of individual objects according to some sorting order without duplicates then we should go for SortedSet.

Methods

```
Object first(); //returns first element of sorted Set
Object  last(); // returns last element of sorted set
SortedSet headSet(Object obj); // returns sortedSet whose elements are less the obj)
SortedSet tailSet(Object obj); // returns sortedSet whose elements are >= the obj)
SortedSet subset(Object obj1, Object obj2)
        // returns SortedSet whose elements are >= obj1 and <obj2
Comparator comparator()
        // returns comparator object that describes underlying sorting technique. If we
```
are using default natural software order then we will get null.

Note

Default natural sorting order

Numbers = Ascending order
String = Alphabetical order

TreeSet (C)

The underlying data Structure is balanced Tree. Duplicate objects are not allowed. Insertion order is not preserved. Heterogeneous objects are not allowed. Null insertion is possible.(only once)

TreeSet implements Serializable and cloneable but not randomAccess.

All objects will be inserted based on some sorting order. It maybe default sorting order or customized sorting order.

Constructors

TreeSet t = new TreeSet(); It creates an empty tree set object where the element will be inserted according to default natural sorting order.

TreeSet t = new TreeSet(Comparator c) creates an empty treeSet object where the elements will be inserted according to customized sorting order, Specified by comparator Object.

TreeSet t = new TreeSet(Collection c);

TreeSet t = new TreeSet(SortedSet s);

Null Acceptance

- For non-empty TreeSet, if we are trying to insert null then we will get null pointer Exception
- For empty TreeSet as a First element null is allowed. But after inserting that null if we are trying to insert any null then we will get runtime exception saying null pointer Exception

Note

Until 1.6V null is allowed for the first element in empty TreeSet. For 1.7V null is not allowed even in the first element. That is null such type of story not applicable for treeSet from 1.7V onwards

TreeSet Rules

If we are depending on default natural sorting order then compulsory the object should be homogenous and comparable otherwise we will get runtime exception saying **ClassCast exception.**

An object is said to comparable if only and only if corresponding class implements comparable interface. String class and wrapper classes already implemented it.

But StringBuffer class doesn't implement comparable interface.

Comparable (I)

It is present in java.lang package and it contains only one method. Which is
public void int compareTo(Object obj)

- Obj1.compareTo(Obj2);
  Returns –ve if obj1 has to come before obj2
  Returns +ve if obj1 has to come after obj2
  Returns 0 if obj1 and obj2 are equal.
  (Obj1 is which is to be inserted. Obj2 is already inserted)

If we are depending on default natural sorting order while adding objects to the TreeSet JVM will call compare to method.

Note

If Default sorting order is not available or if we are not satisfying with default sorting order then we can go for customized sorting using comparator.

Comparator

It is present in java.util package. And it defines two methods which is **compare() and equals().**

Methods

Public int compare(Object obj1 , Object obj2);
Returns –ve if obj1 has to come before obj2
Returns +ve if obj1 has to come after obj2
Returns 0 if obj1 and obj2 are equal.

Public Boolean equals(Object obj);

Note

Wherever we are implementing Comparator Interface. Then compulsory provide implementation only for compare method. But we are not required to provide implementation for equal method. Because it is already available from Object class through inheritance.

Various possible implementation of Compare method

Public int compare(Object obj1 , Object obj2){

Integer i1 = (Integer)obj1;
Integer i2 = (Integer)Obj2;

1. Return i1.compareTo(i2); // Ascending order.
2. Return -i1.compareTo(i2); // Descending order.
3. Return i2.compareTo(i1); // Descending order.
4. Return –i2.compareTo(i1);// Ascending order.
5. Return +1; // Insertion order.
6. Return -1; // reverse of Insertion order.
7. Return 0; // only first element will be inserted and all other element will be considered as duplicate.

Note

If we are depending on default natural sorting order. Then compulsory object should be homogenous and comparable otherwise we will see runtime exception called CCE.

If we are defining our own sorting by comparator then objects need not be comparable and homogenous. That is we can add heterogeneous non- comparable object also.

Comparable VS Comparator

- For pre-define comparable classes default natural sorting order already available. If we are not satisfied with that default order then we can define our own sorting by using comparator.
- For pre-defined non comparable classes like String Buffer default natural sorting order not already available. We can define our own sorting by using comparator.
- Our won classes like Employee, the person who writing class is responsible to define natural sorting order by implementing comparable interface. The person who using our class is not satisfied with Default natural sorting order can define his own sorting by using comparator.
- Comparable implements in **String and All wrapper classes.**
- Comparator implements in **Collator and RuleBasedCollator.**

MAP

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. Each Key – value pair is called Entry, hence Map is considered as collection of Entry Object.

Methods

```
Object put(Object K , Object V)
```

To add one K-V pair to the map. If the key is already present then old value will be replaced with new Value and returns old value.

```
Object putAll(Map m);
Object get(Object K);
Object remove(Object key);
Boolean containsKey(Object Key);
Boolean containsValue(Object Value);
Boolean isEmpty();
int size();
void clear();
```

Collection views of Map

```
Set KeySet(); // Returns all keys in key.
Colllection values(); // returns all values present in map.
Set entrySet(); // Returns both Key and value in amp.
```

Entry (I)

A map is group of key-Value pair. And each Key – Value pair is called Entry. Hence map is considered as Entry Objects. Without existing Map object there is no chance of existing entry object. Hence entry interface is defined inside map interface.

```
Interface Map {
Interface Entry {
        Object getKey();
        Object get Value();
        Object setValue(Object name); // returns Old Object.
    }
}
```

HashMap

The underlying data Structure is HashTable. Insertion order is not preserved and it is based on HashCode of Keys. Duplicates Keys are not allowed and values can be duplicated. Heterogonous objects are allowed for both key and value. Null is allowed for Key only once and for Value any number of times. HashMap implements serializable and cloneable but not Random Access. It is best choice if our frequent operation is search operation.

Constructors

HashMap m = new HashMap(); creates default initial capacity 16. And default filled ratio is 0.75.

HashMap m = new HashMap(int initial Capacity);

HashMap m = new HashMap(int intialCapapcity , float filledRatio);

HashMap m = new HashMap(Map m);

Differences between HashMap and HashTable

- HashMap is not Synchronized and HashTable is Synchronized.
- HashMap is not thread – safe. Whereas HashTable is Thread-Safe.
- Null is applicable for both key and value. (only once for key) where in HashTable Null is not applicable we get **nullPointerException**

Note

By defaukt HashMap is Non-Synchronized, but we can get Synchronized version of HashMap by using **synchronizedMap** method of collection Class.

HashMap m = new HashMap();
Map m1 = **Collection.synchronizedMap(m);**

LinkedHashMap

It is child class of HashMap. It is exactly same as HashMap. Includimg method and constructors except the following difference

- The underlying data structure is HashTable and LinkedList (Hybrid Data Structure)
- Insertion order is preserved.

Note

LinkedHashSet and LinkedHashMap are commonly used for developing cache based Application.

IdentityHashMap

It is exactly same as HashMap including methods and constructors except the following difference. In the case of normal HashMap JVM will use **.equals()** method to identify duplicate Keys Which is meant for content comparison. But in case of Identity HashMap JVM will use == operator to identify duplicate keys which is meant for reference comparison. (Address Comparison)

WeakHashMap

It is exactly same as HashMap except the following difference.

- In the case of HashMap, Even through object doesn't have any reference it is not eligible for GC. If it is associated with HashMap. That is HashMap dominates GC.
- But in the case of WeakHashMap if object doesn't contain any references it is eligible for GC even through object associated with WeakHashMap. That is WeakHAshMap dominates GC.

SortedMap

It is a child interface of Map. If we want to represent a group of Key-Value pair according to sorting orders of keys. Then we should go for sorted map. Sorting is based on key but not based on value.

Methods

Object firstKey();
Object lastKey();
SortedMap headMap(Object Key)
SortedMap tailMap(Object key)
SortedMap subMap(Object key1, object key2)
Comparator comparator();

TreeMap

The underlying data structure is RED – BLACK Tree. Insertion order is not preserved. It is based on some sorting order of Key. Duplicates Keys are not allowed but values can be duplicated. If we are depending default natural sorting order then keys should be homogenous and comparable. Otherwise we will get runtime exception saying **ClassCast exception**. If we are defining our own comparator then the keys need not be homogenous and comparable. We can take heterogeneous and non- comparable object also. Whether we are depending on default natural sorting order or customized sorting order there are no restriction for values. We can take heterogonous non comparable object also.

Null acceptance

For value we can use Null for any number of time. But for key it is not acceptable.

Constructors

TreeMap t = new TreeMap(); //default natural sorting order.
TreeMap t = new TreeMap(Comparator c);// for customized sorting order.
TreeMap t = new TreeMap (SortedMap m);
TreeMap t = new TreeMap (Map m);

HashTable

The underlying data structure is HashTable. Insertion order is not preserved. It is based on HashCode of keys. Duplicate keys are not allowed but values can be duplicated, Null is not allowed for both Key and value. It implements serializable and cloneable interface, but not Random Access. Every method present in HashTable is synchronized hence it is thread – safe. HashTable is best if frequent operation is Search operation. It is heterogeneous.

Constructors

HashTable h = new HashTable(); creates an empty hashTable object with default initial capacity 11 and default fill ratio is 0.75.

HashTable h = new HashTable(int initialCapacity);

HashTable h = new HashTable(int initialCapacity, float fillRatio);

HashTable h = new HashTable(Map m);

Note

The insertion order taken from HashTable is from **Top to bottom and Right to Left.**

Properties

Both key and values will always will be in String Type.

Method

String setProperty(String pname , String pvalue);
String getProperty(String pname);
Enumeration propertyNames();

void load(InputStream is);
void store(OutputStream os , String comment);

Queue (I)

It is a 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 SMS Message we have to store all mobile number in some data Structure. Which we added mobile number same order only message should be delivered. For this FIFO requirement Queue is Best choice,

Usually Queue follows FIFO order but based on our requirement we can implement our own priority order also. From 1.5V onwards LinkedList class also implements Queue Interface.

LinkedList based implementation of Queue always follow FIFO order.

Methods

Boolean offer(Object o) // to add an object into the queue.
Object peek()   // to return head element of the queue. If queue is empty then this method returns null.
Object element() // to return head element of the queue. If queue is empty then this method raises RE: NoSuchElementException
Object poll()// to remove and return head element of the queue. If queue is empty then this method returns null.
Object remove //// to remove and return head element of the queue. If queue is empty then this method raises RE: NoSuchElementException.

PriorityQueue

If we want to represent a group of individual objects prior to processing according to some priority then we should go for priority queue.
The priority can either default sorting order or customized sorting order defined by comparator. Insertion order is not preserved and it is based on some priority,
Duplicate objects are not allowed. If we are depending on default natural sorting order compulsory object should be homogenous and comparable. Otherwise we will get runtime exception saying ClassCastException. If we are defining our own sorting by comparator then objects need not be homogenous and comparable. Null is not allowed even as the first element.

Constructor

PriorityQueue q = new PriorityQueue(); creates an empty PriorityQueue with default initial Capacity 11 and all objects will be inserted according to default natural Sorting order.

PriorityQueue q = new PriorityQueue(int initialCapacity);

PriorityQueue q = new PriorityQueue(int initialCapacity, Comparator C);

PriorityQueue q = new PriorityQueue(Collection c);

Note

Some Platform won't provide proper support for ThreadPriority and PriorityQueue.

NavigableSet (I)

It is the child interface of sortedSet. And it defines several set method for navigation purposes.

Methods

floor(e) // It returns highest element which is <= e.
lower(e) // it returns highest element which is < e.
ceiling(e) // it returns lowest element which is >= e.
higher(e) // it returns lowest element which is >e.
pollFirst() // remove and return first element.
pollLast() // remove and return last element.
descendingSet() //It returns NavigableSet in reverse order.

NavigableMap (I)

It is the child interface of SortedMap. And it defines several map method for navigation purposes.

Methods

floorKey(e) // It returns highest element which is <= e.
lowerKey(e) // it returns highest element which is < e.
ceilingKey(e) // it returns lowest element which is >= e.
higherKey(e) // it returns lowest element which is >e.
pollFirstEntry() // remove and return first element.
pollLastEntry() // remove and return last element.
descendingMap() //It returns NavigableSet in reverse order.

Collections

Collection class defines several utility methods for collection objects like sorting, searching, reversing, etc.

Methods

For sorting elements of List

Public static void sort(List l); // default natural sorting order. While doing sorting the list should be homogenous and comparable. Null is not allowed.

public static void sort(List l , Comparator c); //Customized sorting order.

For searching elements of list

Collections class defines binary Search methods:

Public static int binarySearch(List l , Object target); if the list is sorted according to default natural sorting order then we have to take this method.

Public static int binarySearch(List l , Object target, Comparator c); we have to use this method, if the list is sorted according to customized sorting order.

Conclusion

The above search method internally use binarySearch algorithm. Successful search returns index. Unsuccessful search returns insertion point.

Insertion point

It is a location where we can place target element in the sorted list. Before calling binary search method, compulsory list should be sorted otherwise we will get unpredictable result. If the list is sorted according to comparator then at the time of search operation also we have to pass same comparator object otherwise we will get unpredictable result.

Insertion point start from -1.

Note

For list of n elements

Successful search result rang: 0 to n-1;
Unsuccessful search result range: -(n+1) to -1;
Total result range: -(n+1) to n-1;

For reverse elements of list

Collections class defines following reverse method to elements of list.

Public static void reverse(List l);

Reverse() VS reverseOrder()

- We can use reverse() to reverse the order of element list.
- We can use reverseOrder() to get reversed comparator.

Comparator c1 = collections.reverseOrder(Compaarator c);
c1 is Descending order.
c is Ascending order.

Arrays (C)

Arrays class is a utility class to define several utility methods for array object.

Methods

For sorting elements of Array.

```
Public static void sort(Primitive[] p);
public static void sort(Object[] o);
public static void sort(Object[] o, Comparator c);
```

Note

We can sort primitive Array only based on D.N.S.O. we cant sort Object[] either based on default natural sorting and customized sorting order.

For searching elements of Array.

```
Public static int binarysearch(primitive[] p, primitive target);
Public static int binarysearch(Object[] o, Object target);
Public static int binarysearch(Object[] o, Object target, Comparator c);
```

Note

All rules of Arrays class binarySearch method exactly same as Collection class BinarySearch.

Conversion of Array to list

Public static List asList(Object[] o); strictly speaking this method won't create a independent List object. For the existing array we are getting List view.

By using array reference if we perform any change automatically that change will be reflected for the List.

By using list reference if we perform any change that change will be reflected automatically to the array the array

By using list reference we can't perform any operation which varies the size otherwise we will get runtime exception saying **UnsupportedOperationExeption.**

By using list reference we are not allowed to replace with heterogeneous object otherwise we will get runtime exception saying **ArrayStoreException.**