***An array is an index collection of fixed nos. of homogeneous data elements.***

***The main advantage of array is we can represent multiple value by using single variable so that readability of code is improved.***

***LIMITATION OF ARRAYS:***

* ***Arrays are fixed in size i.e. once we create an array there is no chance of increasing or decreasing size based on our requirement due to this to use array concepts compulsory, we know the size in advance. Which may not possible always.***
* ***Array can hold only homogeneous Data type elements exam.***

***Student [] s=new Student [1000];***

***S [0] = new Student ();***

***S [0] = new customer (); \*\*\*this will show an error***

***We can solve this problem by object type Array.***

***Object [] s=new Object [1000];***

***S [0] = new Student ();***

***S [0] = new customer () ;/\*\*\*\*both are valid.***

* ***Arrays concept is not implemented based on some std. Data structure and hence readymade method support is not available for every requirement we have to write code explicitly, which increases complexity of programming.***

***TO overcome above problems of array we should go for COLLECTION CONCEPT***

* ***Collections are growable in nature, i.e. based on our requirement we can increase or decrease the size.***
* ***Collection can hold both homogeneous and heterogeneous objects.***
* ***Every collection class is implemented based on some std. data structure hence for every requirement ready made method support is available being a programmer we are responsible to use those method and we are not responsible to implement those methods.***

***Differences between Arrays & collection.***

|  |  |
| --- | --- |
| ***Arrays*** | ***Collection*** |
| ***Arrays is fixed in size, once we create an array we can’t increase and decrease size based on our requirement.*** | ***Based on our requirement we can increase and decrease size*** |
| ***With respected to memory Array is not recommended to use*** | ***w.r.t memory it is recommended to use.*** |
| ***w.r.t performance Arrays is recommended to use*** | ***w.r.t performance collection are not recommended to use*** |
| ***It can hold only homo. Data type element*** | ***Collection can hold both homo, hetro elements.*** |
| ***There is no underline data structure for Arrays and hence ready made method support is not available, for every requirement we need to write explicitly that increase complexity*** | ***Every collection class is implemented based on some std. data structure and hence for every requirement ready made method support is available being a programmer we can used this method directly*** |
| ***Arrays can hold both primitives and the objects*** | ***It can hold only object 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, Interfaces which can be used to represent a group of individual obj. as a single entity.***

***9 key interfaces of collection Framework:***

1. ***Collection***
2. ***List***
3. ***Set***
4. ***Sorted Set***
5. ***Navigable Set***
6. ***Queue***
7. ***Map***
8. ***Sorted Map***
9. ***Navigable Map***

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

***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 collection.***

***Collections is a utility class present in JAVA.UTIL package to define several utility methods for collection objects [like sorting, searching, etc.]***

***2) LIST {I}:***

* ***It is the child interface of collection***
* ***If we want to represent group of individual objects as a single entity***

***Where duplicates are allowed and insertion ordered should be preserved then we should go for LIST.***

***3) set (I):***

* ***It is child interface of collection.***
* ***If we want to represent group of individual objects as a single entity***

***Where duplicates are not allowed and insertion ordered not required to be preserved then we should go for SET.***

***4) Sorted set (I):***

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

***5) navigable Set (I):***

* ***It is child interface of sorted set***
* ***It contain several methods for navigation purposes.***
* ***For diagram go to register .***

***6) queue (I):***

* ***It is the child interface of collection***
* ***If we want to represent a group of individual obj. prior to processing then we should go for queue.***
* ***Usually queue follows 1st in 1st out order but based on our requirement we can implement our own priority order also.***
* ***Eg. Before sending a mail all mail ID’s we have to store in some data structure, in which order we added mail ID’s, in the same order only mail should be delivered for this requirement Queue is the best choice.***

***NOTE: all the above interfacess ( collection,list,sorted set,navigable set and queue) meant for representing a group of individual objects.***

***If we want to represent a group of object as KEY-VALUE-PAIRES then we should go for MAP concept.***

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

* ***Map is not child interface of collection.***
* ***If we want to represent a group of objects as KEY-VALUES paires then we should go for map.***

|  |  |
| --- | --- |
| ***KEY*** | ***VALUE*** |
| ***1*** | ***Bushra*** |
| ***2*** | ***zoha*** |
| ***3*** | ***faizan*** |

1. ***Both key and value are objects only.***
2. ***Duplicate keys are not allowed but value can be duplicated.***

***8) Sorted map:***

* ***It is the child interface of map***
* ***If we want a represent group of key value pairs according to some sorting order of keys, then we should we go for sorted map.***
* ***In sorted map the sorting should be based on key but not based on value.***

***9) navigable map:***

***It defines several methods for navigation purposes.***

***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

***1) COLLECTION(I):***

1. ***If we want to represent a group of individual object as single entity then we should go for collection.***
2. ***Collection interface defines the most common method which are applicable for any collection object.***

***Different methods of collection interface which is applicable to all its child interface.***

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 except those who in c***
6. ***void clear()***
7. ***boolean contains(Object 0 ) object is present or not***
8. ***boolean containsAll(Collecion c)***
9. ***Boolean isEmpty()***
10. ***Int size() to check how many***
11. ***Object [ ] ---- toArray();***
12. ***Iterator -----Iterator() if we want object one by one***

***NOTE: there is no concrete class which implement inter. Directly***

1. ***List( I): child intereface of collection (I),duplication are allowed and insertion ordered must be preserved then we should go for List.***

***We can preserve insertion order via index & we can differentiate duplicate object by using index hence index will play very important role in List***

***List interface defines following methods***

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 at specific index with provided object and return old object***
6. ***int indexOf(object o)***

***return index of fist occurrence of “O”***

1. ***int lastIndexOf(object o)***

***return index of last occurrence of “O”***

1. ***ListIterator listIterator();***

***Four classes in list(I):***

1. ***ArrayList***
2. ***LinkedList***
3. ***Vector ¬stack***

***ArrayList: underline data structure is resizable array or cc growable array***

***Duplicates are allowed***

***Insertion order is preserved***

***Heterogeneous object are allowed ( except TreeSet and TreeMap) everywhere hetro. Object are allowed) why?***

***Because TreeSet and TreeMap implements the SortedSet and sortedMap – It means that it uses the comparator internally to store the elements based on some sorting order and to achieve that we need all element should be same.***

***Null insertion is possible.***

***CONSTRUCTOR:***

1. ***ArrayList A=new ArrayList();***

***It will creates empty ArrayList object with default initial capacity “10”. Once ArrayList reaches its max capacity a new ArrayList object will be created with new capacity is equal to.***

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

1. ***ArrayList A = new ArrayList(int initialcapacity);***

***It will create empty ArrayList object with specified initial capacity***

1. ***ArrayList A = new ArrayList(Collection c);***

***It will create equivalent ArrayList object for given collection ( treeset,stack etc)***

***Note: usually we can use collections to hold and transfer object from one location to another location{container}.o provide support for this requirement every collection class by default implements serializable and cloneable interfaces***

***Arraylist and vector classes implement RandomAccess interface so any random element we can access with the same speed***

***RndomeAcess Interface : present in JAVA.Util.Package and it doesn’t contain any methods. It is a Marker interface where required ability will be provided automatically by the JVM***

1. ***If our frequent operation is retrieval then ArrayList is the best option ( bcz it supports RandomAcess Interface)***
2. ***ArrayList is the worst choice if our frequent operation is Insertion or deletion in the middle.***

***Differences between ArrayList and the vector***

|  |  |
| --- | --- |
| ***ArrayList*** | ***Vector*** |
| ***Every method is non synchronized*** | ***Every method present in vector is synchronized*** |
| ***At a time Multiple thread are available to operate and hence it is not thread safe*** | ***At a time only one thread is allowed to operate on vector object and hence it is thread safe*** |
| ***Relatively performance is high bcz threads are not required to wait to operate in ArrayList object*** | ***Performance is low relatively bcz threads need to wait*** |
| ***1.2 V*** | ***1.0 legacy*** |

***Q 1) how to get synchronized version of ArrayList object?***

***By default ArrayList is non synchronized but we can get synchronized version of ArrayList object by method of collections class***

***(Public Static List synchronizedList(List L))***

***E.G:***

***ArrayList L = new ArrayList();***

***List l1 = Collections.synchronizedList (l);***

***Similarly we can get synchronized version of Set and Map object by using following methods of collections class***

***Public Static Set synchronizedSet(Set s)***

***Public Static Map synchronizedMap(Map m)***

***LinkedList:***

1. ***Underline data structure is double linked list***
2. ***Insertion order is preserved***
3. ***Duplicate objects are allowed***
4. ***Hetro. object are allowed***
5. ***Null insertion is possible***
6. ***It implements Serializable and Colonel interface but not Random access interface.***
7. ***It is best choice when frequent operation is insertion and deletion in middle***
8. ***It is worst choice if our frequent operation is retrieval operation.***

***Constructors:***

1. ***LinkedList L=new LinkedList();***

***Creates empty list object***

1. ***LinkedList L=new LinkedList(Collection C);***

***Create equivalent LinkedList object for given collection.***

***LinkedList class contains following method***

***Usually we can use LinkedList to develop stack and queue, to provide support this requirement LinkedList class defines the following methods.***

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

***Differencess between ArrayList and LinkedList***

|  |  |
| --- | --- |
| ***ArrayList*** | ***LinkedList*** |
| ***It is the best choice when frequent operation is retrieval operation*** | ***It is the best choice if frequent operation is insertion and deletion in middle*** |
| ***It is worst when insertion or deletion in middle bcz internally several “shift” operation is performed*** | ***It is worst when retrieval operation*** |
| ***Memory stored in consequent memory location*** | ***Vice versa*** |

***Vector:***

1. ***Underline data structure is resizable array or growable array.***
2. ***Insertion is preserved***
3. ***Duplication are allowed***
4. ***Heterogeneous objects are allowed***
5. ***Null insertion is possible***
6. ***It implements serializable, cloneable and Random-access interface***
7. ***Every method present in vector is synchronized and hence it is safe in thread***

***Constructor:***

1. ***Vector v=new Vector();***

***Creates with default capacity 10***

***Once it reaches its max capacity then new vector object will be created with (new capacity = current capacity \*2)***

1. ***Vector v = new Vector(int intialcapacity)***

***Creates empty vector with specified capacity.***

1. ***Vector v =new Vector(int initialcapacity, incremental capacity)***
2. ***Vector v =new Vector(Collection c)***

***Collect equivalent vector object for given collection, this constructor meant for interconversion between objects)***

***Vector specific method:***

***{ add(object o)-----colle.***

***add(int index,Object o)----list***

***addElement(object o)-----vector }***

***for remove {***

***remove(object o)-----C***

***removeElement(object o)---V***

***remove(int index)---L***

***removeElemetAt(int index)—V***

***clear()---C***

***removeAllElement()---V }***

***to get objects{***

***Object get(int index)---L***

***Object elementAt(int index)—V***

***Object firstElement()---V***

***Object lastElement()---V}***

***Other miscellaneous method of vector***

***Int size()***

***Int capacity()***

***Enumeration element()***

***Stack:***

***Child class of Vector, especially design for LIFO***

***Only one default constructor***

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

***Methods:***

1. ***Object push(Object 0)***

***To insert object into stack***

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()***

***Return true if the stack is empty***

1. ***Int search(object 0)***

***Return offset if the element is available otherwise return -1***

***THREE CURSERS OF JAVA:***

***If we want to get an object one by one from the collection then we should go for curser.***

***There are three types of curser available in java***

1. ***Enumeration***
2. ***Iterator***
3. ***Listiterator***
4. ***Enumeration:***

***We can use this to get object one by one from legacy collection object***

1. ***We can create enumeration //\*\*object by using “Elements method” of vector class .***

***Public Enumeration elements();***

***e.g:***

***Enumeration e =v.elements();***

***Where v is vector object***

***Methods of enumeration:***

1. ***Public Boolean hasMoreElements();***
2. ***Public Object nextElement();***

***Limitation of enumeration:***

1. ***We can apply enumeration concept only for legacy classes and it’s a not a universal curser***
2. ***By using enumeration, we can get only read access and we can’t perform remove operation***
3. ***To overcome above limitation***
4. ***We should go for iterator***

***Iterator***

1. ***We can apply iterator concept for any collection object and hence it is a universal curser***
2. ***By using iterator, we can perform both read and remove operation***
3. ***We can create iterator //\*\*object by using iterator method of collection interface***

***Public Iterator iterator()***

***e.g:***

***Iterator itr=c.iterator();***

***C is any collection object***

***Methods:***

1. ***Public Boolean hasNext()***
2. ***Public object next()***
3. ***Public void remove()***

***Limitation of iterator:***

1. ***We can always move in forward direction we cant move towards backward direction bcz these are single direction curser but not the bi directional curser both (E,I )***
2. ***We can only perform read and remove operation and we cant perform replacement and add operation***
3. ***To overcome above limitation we should go for ListIterator***

***ListIterator:***

1. ***By using it we can move in both direction bcz it is bi direction curser***
2. ***We can perform Read, Remove, Replace, Add operation***

***We can create list iterator by using listIterator method of list interface***

***Public ListIterator listiterator()***

***e.g:***

***ListIterator ltr=l.listIterator();***

1. ***Listiterator is chiuld interface of Iterator (i)***
2. ***And hence all method present in iterator by default available to the listIterator***

***9 methods in ListIterator:***

Forward

1. ***Public Boolean hasnext()***
2. ***Public object next()***
3. ***Public int nextIndex()***
4. ***Public Boolean hasPrevious()***

Back ward

1. ***Public object previous()***
2. ***Public int previousIndex()***
3. ***Public void remove()***

Extra op.

1. ***public void add(object)***
2. ***public void set(object)***

***limitation:***

***it is applicable to list object only.***

***COMPARISION TABLE OF 3 CURSER***

|  |  |  |  |
| --- | --- | --- | --- |
| ***Property*** | ***Enumeration*** | ***Iterator*** | ***ListIterator*** |
| ***Where we can perform*** | ***Only for legacy classes*** | ***For any collection object*** | ***Only for list object*** |
| ***It is legacy*** | ***Yes(1.0)*** | ***No(1.2)*** | ***No(1.2)*** |
| ***Movement*** | ***Single direction(F)*** | ***Single direction (F)*** | ***Bi direction*** |
| ***Allowed operation*** | ***Only read*** | ***Read and remove*** | ***Red, remove, add, replace*** |
| ***How we can get*** | ***Element()***  ***Of vector class*** | ***Iterator()***  ***Of collection(I)*** | ***listIterator()***  ***of List(I)*** |
| ***Methods*** | ***2 method*** | ***3 method*** | ***9 method*** |

***When explaining to an interviewer why the `remove` and `add` operations on an `Iterator` do not throw a `ConcurrentModificationException`,*** *you can use the following explanation:*

*"The `remove` and `add` operations on an `Iterator` do not throw a `ConcurrentModificationException` because these iterators are designed to be fail-fast iterators, meaning they are capable of handling modifications to the underlying collection while iterating. This behavior is not a contradiction; it's by design.*

*Here's why it works this way:*

*1. `remove`: When you call `iterator.remove()`, the iterator knows exactly which element it is currently pointing to in the collection. It can safely remove that element from the collection because it is under the iterator's control. The iterator is aware of its own modifications, so it can adjust its internal state accordingly.*

*2. `add`: When you call `iterator.add(element)`, the iterator is responsible for inserting the new element into the collection at the appropriate position. It does this without interfering with the ongoing iteration because it maintains its own position within the collection.*

*In essence, the iterator is designed to be self-aware. It can manage its own modifications to the collection without causing inconsistencies or concurrency issues. However, if you were to modify the collection directly while iterating with an iterator, such as using the `add` or `remove` methods of the collection itself, that would indeed lead to a `ConcurrentModificationException`. This exception is intended to catch situations where the collection is modified by external factors during iteration, helping developers detect potential concurrency problems.*

*In summary, the behavior of `remove` and `add` methods on an `Iterator` is not a loophole but a well-thought-out design to allow controlled modifications while iterating without risking data corruption or undefined behavior."*

*This explanation should demonstrate your understanding of the concept and convince the interviewer that you have a solid grasp of how iterators work in Java.*

***SET:***

***Set is child interface of collection, if we want to represent a group of individual object as a single entity “where duplications are not allowed and insertion order not preserved “then we should go for set.***

***Set interface doesn’t contain any new method and we have to use only collection interface method.***

***HashSet:***

1. ***underline data structure is Hash table***
2. ***duplicate objects are not allowed***
3. ***insertion order is not preserved***
4. ***and its based-on hash code of object***
5. ***null insertion is possible (only once)***
6. ***hetro objects are allowed***
7. ***implements serializable and clonoble but not marker interface (RAI)***
8. ***best choice when frequent operation is search operation.***

***Note: in hash set duplication is not allowed if we are trying to insert duplicate then we won’t get any compile time and run time error and add method simply return false (Boolean).***

***Constructor:***

1. ***HashSet h=new HashSet();***

***It will create empty hashset creates empty object with default initial capacity 16 and the “ default fill ratio=0.75” (load factor)***

1. ***HashSet h =new HashSet(int initialcapacity)***

***Creates with the specified initial capacity and the default fill ratio is “0.75”***

1. ***HashSet h=new HashSet(int initialcapacity,float fillRatio);***
2. ***HashSet h=new HashSet(collection c);***

***Creates equivalent HashSet for given collection ,this constructor meant for inter-conversion between collection object.***

***NOTE:***

***Fill Rtio/Load factor:***

***After filling how much ratio new hashset object will be created this ratio is called fill ratio or load factor.***

***Ex. Fill ratio=0.75 means after filling 75% ratio a new hashset object will be created.***

***LinkedHashSet:***

***It is the child class of HashSet, it is exactly same as HashSet including Constructor and Method except following differences.***

|  |  |
| --- | --- |
| ***HashSet*** | ***LinkHashSet*** |
| ***Data structure is Hash table*** | ***Data structure is a combination of LinkedList and HashTable*** |
| ***Insertion order not preserved*** | ***Insertion order preserved*** |
| ***1.2 v*** | ***1.4*** |

***In the above program if we replace HashSet with LinkedHashSet then out put is according to above .***

***Note: in general we can used LinkedHashSet to develop cache based Application where duplications are not allowed and insertion order preserved***

***SortedSet(I):***

***Is the child interface of set where duplication's are not allowed and insertion order must be according to some sorting order.***

***Sorted interface defines the following specific methods:***

1. ***Object first();***

***Return first element of the sortedSet***

1. ***Object last();***

***Return last element of the SortedSet***

1. ***SortedSet headset(object obj)***

***Return SortedSet whose element are less than obj***

1. ***SortedSet tailSet(Object o)***

***Return SortedSet whose element are>=obj***

1. ***SortedSet subset(object obj1,object obj2)***

***Return elements are>=obj1 and <obj2***

1. ***Comparator comparator()***

***Return object that describe underline sorting technique. If we are using default sorting it will return Null.***

***For nos ----Ascending order***

***For String ----Alphabetical order***

***This are the above default natural sorting order.***

***E.x – 100,101,104,106,110,115,120***

***Answer according to above method serial wise***

***100,120,[100,101,104],[106,110,115,120],[101,104,106,110],Null***

***TreeSet(implementation class of sortedset):***

1. ***Underline data structure is balanced tree***
2. ***Duplicates are not allowed***
3. ***Insertion order not preserved***
4. ***Hetro. objects are not allowed otherwise we will get run time exception [ when we used default]***
5. ***Null insertion possible only once***
6. ***It implements serializable and cloneable but not Random Access***
7. ***All objects should be inserted based on some sorting order it may be default sorting order or customize sorting order***

***Constructor:***

1. ***TreesSet t=new TreeSet();***

***Where elements is inserted according to default natural order***

1. ***TreesSet t=new TreeSet(Comparator);***

***Where elements are inserted according to customize sorting order specified by comparator object.***

***3) TreesSet t=new TreeSet(Collection c);***

***4) TreesSet t=new TreeSet(Sorted s);***

***Note:***

***For non empty TreeSet if we are trying to insert Null we will get Null Pointer Exception***

***For empty TreeSet Null is allowed but after inserting null if we are trying any other object then we will get run time exception.***

***\*\*until 1.6 version was allowed as first element as empty tree set but 1.7 version after it is not allowed, so Null such type of story is not applicable after 1.7v***

***If we are using default constructor the object should be homogeneous and comparable otherwise it will show exception.***

***If corresponding object class implements “comparable interface” then it is comparable.***

***Comparable (I):***

***It contain only one method and present in Java.lang package .***

***compareTo();***

***public int compareTo(Object obj);***

***ex. Obj1.compareTo(obj2)***

1. ***Return -ve if obje1 has to come before obj2***
2. ***Return +ve if obj1 has to come after obj2***
3. ***Return 0 if obj1 and obj2 both are equal n***

***If we are depending on default natural sorting order then while adding object in to Treeset JVM call compareTo method.*** We can say that TreeSet<T> uses the Comparable internally for natural sorting order.

***Ex. TreeSet t= new TressSet();***

***t.add(“k”);--first object***

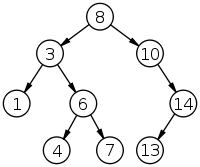
***t.add(“a”);--“a” compareTo(“k”); -ve***

***t.add(“a”);---" a” compareTo(“a”); 0***

***t.add(“z”); “z” compareTo(“k”);+ve***

***syso(t)***

***o/p [a,k,z]***



[This Photo](http://stackoverflow.com/questions/17683803/java-treeset-ordering-algorithm) by Unknown Author is licensed under [CC BY-SA](https://creativecommons.org/licenses/by-sa/3.0/)

***Note : if natural default sorting order is not available or if we are not satisfied with the default natural sorting order then we can go for customized sorting by using comparator***

***Comparable meant for Default natural sorting***

***Order***

***Comparator meant for customized sorting order***

***Note: comparable present in java.lang package***

***And comparator present in java.util package***

***Comparator: two method***

***1) Public int compare(obj1,obj2)***

1. ***Return -ve if obje1 has to come before obj2***
2. ***Return +ve if obj1 has to come after obj2***
3. ***Return 0 if obj1 and obj2 both are equal n***

***2] public Boolean equals (object obj)***

***Whenever we are implementing comparator interface compulsory, we should provide implementation only for compare method but we are not providing it for equals method.***

***Bcz its already available from object through inheritance.***

***Question 1) write a program to insert INTEGER object to TreeSet where sorting order is descending oder.***

Important point : When u are adding value in tree set then internally it uses the comparable<Used Type>

so if custom object we are trying to add then custom object should implements the Comparable<MyClass>.

***Various possible implementation of compare method:***

***For better understanding refer set package***

**static** **class** My\_Comparator1 **implements** Comparator<Object> {

**public** **int** compare(Object obj1, Object obj2) {

// **TODO** Auto-generated method stub

String i1=(String)obj1;

String i2=(String)obj2;

1. ***Return i1.compareTo(i2)\*\*\*\*default sorting method***
2. ***Return – i1.compareTo(i2)\*\*\*\*Descending order***
3. ***Return i2.compareTo(i1)\*\*\*descending order***
4. ***Return -i2.compareTo(i1)\*\*\*\*Ascending order***
5. ***Return +1; [ comes according to insertion order]***
6. ***Return -1;[ reverse of insertion order]***
7. ***Return 0;[only first element will come then all duplicates that’s why will dive only 1st object only]***

***NOTE:***

***If we depend on natural sorting order then object should be comparable and it should be homogeneous. Otherwise we will get runtime exception.***

***If we are defining our own sorting by comparator then objects need not be comparable and not to be homogeneous.***

***If two objects having same length then consider there alphabetical order***

***Comparable vs comparator***

1. ***For pre determine comparable classes deafult sorting order is already available if we are not satisfied with default sorting then we can define over own sorting by comparator***
2. ***For some classes default sorting is not available so we can define our own sorting by comparator***
3. ***For our classes like Employee, the person who is writing the class is responsible to define default sorting order by comparable interface***

***If the person who is not satisfied by natural sorting order then he can define by using comparator .***

***Comparison of comparable and comparator***

|  |  |
| --- | --- |
| ***Comparable*** | ***Comparator*** |
| ***It is for default sorting order*** | ***It is for customizer sorting order*** |
| ***Java.lang*** | ***Java.util*** |
| ***Only one method compareTo()*** | ***Two method***  ***Compare()***  ***Equal()*** |
| ***All string and rapper classes*** | ***Implemented classes is collector and RuledBased Collector*** |

***Comparison table of Set implemented classes***

|  |  |  |  |
| --- | --- | --- | --- |
| ***Point*** | ***HastSet*** | ***LinkedHashSet*** | ***TreeSet*** |
| ***Data structure*** | ***Hashtable*** | ***Linkedlist and hashtable*** | ***Balanced tree*** |
| ***Duplicates*** | ***No*** | ***No*** | ***No*** |
| ***Insertion order*** | ***No*** | ***Preserved*** | ***No*** |
| ***Sorting*** | ***Not appli.*** | ***No.*** | ***Applicable*** |
| ***Hetro*** | ***Allowed*** | ***Allowed*** | ***Not(default)*** |
| ***Null*** | ***Allowed*** | ***Allowed*** | ***No such case*** |

***Map(I):***

***1) map is not child interface of collection.***

***2) if we want to represent as key value paires then we should go for Map***

|  |  |
| --- | --- |
| **101** | **Shifa** |
| **102** | **Kahekahs** |
| **103** | **Faizan** |

1. **Both key and values are objects only**
2. **Duplication keys are not allowed but value can be duplicated**
3. **Each key value pair is called Entry. hence map is considered as a collection of entry objects**

**Map Interface Methods:**

1. **Object put(Object key , Object value):**

**To add one key value to the map**

**If the key is already present then old value will be replaced with new value and returns old value**

**Ex. M.put( 101,”shiva”);//return Null \*\*\*\*\*replaced new value by Anisha**

**M.put(102,”ravi”);Null**

**M.put(103,”ravina”);//null**

**M.put(101,”anisha”);go up and return old object value shiva**

1. **Void putAll(Map m);**
2. **Object get(Object key) returns the value associated with specified key**
3. **Object remove(Object key)**
4. **boolean containsKey(Object key)**
5. **boolean containsValue(Object value)**
6. **boolean isEmpty();**
7. **Int size()**
8. **Void clear();**

**Collection views of Map:**

1. **Set keyset()//only give key**
2. **Collection values()**
3. **Set entrySet()**

**Entry(I): A map is a group of key value pairs and each key value pairs is called Entry.**

**Therefore, map is considered as collection of Entry Objects. Without existing map object there is no chance of existing Entry object. hence interface is defined inside Map Interface**

**Intetrface Map{**

**Interface Entry{**

**Object getKey();**

**Object getValue();  
Object setValue(Object obj);return old value**

**}**

**}**

**These three method is specific entry method and we can apply only on Entry object.**

**HashMap:**

1. **Data structure is hash table**
2. **Insertion order is not preserved and it is based on hash code of key**
3. **Duplication key is not allowed but value can be duplicated**
4. **Hetro objects are allowed for both key and value**
5. **Null is allowed for key but only once time but in value we can used it multiple times**
6. **HahMap implements serializable and clonable interfaces but not random interface,**
7. **hash map preferred is frequent operation is search operation**

**Constructor:**

1. **HashMap m= new HashMap()**

**With default initial capacity Is 16 with fill ratio is 0.75**

1. **HashMap m= new HashMap(int initialcapacity);with fill ratio 0.75**
2. **HashMap m= new HashMap(int initialcapacity,float fill ratio)**
3. **HashMap m= new HashMap(Map m)**

**Differences of hashMap and HashTable**

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| **Non synchronized** | **Synchronized** |
| **Thread not safe** | **Thread safe** |
| **1.2** | **1.0 legacy** |
| **Null we can add** | **No such story** |
| **Performance is high** | **Not high** |

**How to get synchronized version of hash map object**

**By default HashMap is non synchronized but we can get synchronized version of hash map by using synchronized map method of collection Class**

**HashMap m=new HashMap();**

**Map m1=Collections.synchronizedMap(m);**

**Where m=non synchronized**

**And m1=synchronized**

**LinkedHashMap:**

**Same difference like hashSet and linkedHashSet**

**Including and method and constructor**

**Difference between == and .equal()**

**In general == meant for reference comparison and .equal for content comparison**

**Integer I= new Integer(10);**

**Integer i1=new Integer (10);**

1. **Sopln(I==I1);----false**

**Bothe are different and represent individual different object having same value-----so it will return false  
2) sopln(I.equals(I1);-----true  
 bcz it deals with content if content is same then will return true**

**IdentityHashMap:**

**It is exactly same as hashmap including method and constructor except following difference**

**In the case of normal hash map JVM will “used .equals” method to identify duplication but it in the case of identihashmap JVM will used “==” to identify duplication which will give u false so here it will show the same key**

**WeakHashMap: it is exactly same as hashmap but have one difference**

**In the case of hasmap even though object doesn’t have any reference variable it is not available for gc if it associated with the hashmap i.e hashmap dominate GC**

**But in the case of weakhasmap if obj doesn’t contain any reference then it is available for GC i.e GC dominates weakHahMap.**

**SortedMap (I): 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.**

**Sorting is based on the key not on value**

**Sorted map defined same method as sortedSet in place of set use map only.**

**Treemap:**

1. **Data structure is RED-BLACK Tree**
2. **Insertion order is not preserved and it is based on some order of keys.**
3. **Duplicate keys are not allowed but value can be.**
4. **If we are depending on default natural sorting order then keys should be homo. And comparable otherwise we will get error.**
5. **If we are defining our own sorting by comparator then keys need not be comparable and homogenous .[customizer sorting order]**
6. **Null Acceptance :**

**Same as like as sortingset.**

**For values we can use null any numbers of time there is no restriction whether it is1.6 or1.7 v**

**Constructor:**

**Same as like treeset**

**HashTable:**

1. **Underline data structure is HasTable**
2. **Insertion order is not preserved and it is based on hash code of key**
3. **Duplication keys are not allowed and values can be duplicated**
4. **Hetro object allowed for both keys and value**
5. **Null is not allowed for both keys and value otherwise we will get null pointer exception**
6. **It implements sereializable and clonoble interface but not random access**
7. **Every method is synchronized hence hastable object is thread safe**
8. **It is best one when frequent operation is search operation.**

**Constructor:**

**Hashtable h = new Hashtable ();**

**With initial default capacity is 11 and 0.75 fill ratio**

**Hashtable h = new Hashtable (int initialcapacity);**

**Hashtable h = new Hashtable (int initialcapacity,float fillRatio);**

**Hashtable h = new Hashtable (Map m);**

**Properties (MAP implemented class): java.util.properties**

**In our program anything which changes so frequently like mail id, mob no, and password. Are not recommend to use while writing a code(hardcode) bcz the value will change w.r.t time.**

**Bcz if we do any change then to reflect that change recompilation ,rebuild and re deploy and sometime server restart will require , which is not at all recommended.**

**So we can overcome with this problem by using properties file.**

**Such type of variable things we have to configure in the properties file . from the properties file we have to read in to java file and use those properties. The main advantage of this, is to use minimum time to reflect that change**

**Bcz for that redeployment is only needed.**

**We can use java properties object to hold properties which are coming from properties file.**

**In normal map implemented classes, key and value can be of any type but in case of properties KEY AND VALUE SHOULD BE “STRING TYPE”.**

**Constructor:**

**Properties p= new Properties( );**

**Methods of properties:**

1. **String setProperty(String Pname, String Pvalue)**

**To set a new property.**

1. **String getProperty(String Pname);**

**To get value associated with the specific property.**

1. **Enumeration propertyNames()**

Properties file

ABC.properties

Load ( );

**prop f b**

Store( );

**void load (InputStream is)**

**to load properties from properties file into java properties object**

**void store(OutputStream os, String comment)**

**to store properties from java properties object into properties file.**

**QUEUE (i):**

**1.5 version enhancements (queue interface )**

**It is the child interface of collection.**

**IF we want to represent a group of individual obj. PRIOR to processing then we should go for queue.**

**For ex. Before sending sms all no we have to store all mobile no in some data structure. So in which order we have stored mob no in same order msg should be deliver. For this 1st in 1st out requirement, then queue is the best choice . usually queue follows 1st in 1st out but based on our requirement we can implement our own priority with the help of priority queue.**

**For 1.5 v LinkedList class also can implement for queue and which will followed 1st in 1st out.**

**Methods queue(I):**



1. **Boolean offer(object o) // to add object into queue**
2. **Object peek( ) //to return head element of the queue.**

**\*\*if the queue element is empty then this method simply returns Null.**

1. **Object element( ) // to return head element of the queue \*\* if queue is empty then it will rise run time exception [ NoSuchElementException]**
2. **Object poll( )// to remove and return head element of the queue.\*\*if there Is no element in queue the return null**
3. **Object remove( )// to remove and return head element of the queue . if no element present in queue then raise an exception [NoSuchElementException].**

**PriorityQueue(class):**

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

1. **The priority can be according to default sorting order or by customize sorting order**
2. **Insertion order is not preserved and it is based on some priority.**
3. **Duplication's are not allowed.**
4. **If we are doing according default sorting order then object should be homo, and it should implement comparable interface.**
5. **For customize, object need not to be comparable**

**And we can add hetro obj.**

**Constructors:**

1. **PriorityQueue q=new PriorityQueue ( );**

**//intial capacity 11, default sorting order.**

1. **PriorityQueue q=new PriorityQueue (int initialcapacity);**
2. **PriorityQueue q=new PriorityQueue(int initialcapacity, Comparator c)//for only customizer sorting order we don’t have separate constructor here , we have to do that with initial capacity.**
3. **PriorityQueue q = new PriorityQueue(sortedSet s);**
4. **PriorityQueue q= new PriorityQueue(Collection c);**

**For example just refer to the IDE in Queue package.**

**Note: some platform won’t provide proper support for thread priority and priority queues.**

**1.6 V ENHANCEMENT IN COLLECTION FRAMEWORK**

1. **NavigableSet(I)**
2. **NavigableMap(I)**
3. **NavigableSet(I) 1.6 v**

**Child interface of sortedSet and it defines the several method for navigation purposes.**

|  |
| --- |
| **00.20** |
| **1.30** |
| **2.22** |
| **3.00** |
| **7.00** |
| **8.52** |
| **11.30** |
| **12.30** |
| **13.25** |
| **17.32** |

**This is the chart of flights boarding**

**From Nagpur to Mumbai.**

***Methods:***

1. ***Floor(e)***

***It returns highest element which is <=e***

***Ex. floor (3.00)//3.00***

***floor (4.00)//3.00***

1. ***Lower(e)***

***It returns element which is < e***

***Ex. lower (3.00) // 2.22***

***lower (4.00) //3.00***

1. ***Ceiling ( e)***

***It returns lowest element which is >= e***

***Ex. ceiling (7.00) // 7.00***

***Ceiling (7.30) // 8.52***

1. ***higher (e)***

***It returns lowest element which is > e***

***higher (7.00)//8.52***

***higher (7.30)// 8.52***

1. ***pollFirst( )***

***remove and return first element***

1. ***pollLast()***

***remove and return last element***

1. ***descendingSet( )***

***it returns NavigableSet in reverse order.***

***TreeSet(C ) implementation class of NvigableSet***

***NavigableMap(I):***

***Child interface of sortedMap and havig different methods for navigable purpose.***

1. ***It defines the following methods.***

***Same methods of above navigableSet but here key we have to put as a parameter.***

***Collections class (utility class for collections objects)***

***define several utility method for collection classes.***

***For sorting of list object:***

1. ***Sorting element of list:***

***Collections class defines 2 sort method.***

**Public static void sort(list l) *–***

***to sort according to default sorting order [ list should compulsory contains homogeneous and comparable object otherwise we will get run time exception classCastException, and it should not contain Null bcz of compareTo( )***

1. ***Customize Sorting of list:***

***Public static void sort(list l.Comparator c);***

***Searching for list object:***

***Collections defines binarysearch method for that***

1. ***Public static int binarySearch(list l,Object target);***

***If the list is sorted according to default sorting order. Then we can use above method.***

1. ***Public static int binarySearch(list l,Object target,Comparator c);***

***If the list is sorted according to customize sorting order. Then we can use above method.***

***NOTE:***

1. ***Above search method internally will use binary search Algorithm.***
2. ***Successful search return int index***
3. ***Unsuccessful return insertion point***
4. ***Insertion point: Where we can place target element in sorted list***
5. ***Before calling above method, list object should be sorted otherwise, we won’t get error but we get unpredicted answer.***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***Z*** | ***A*** | ***M*** | ***K*** | ***B*** | ***G*** | ***S*** |

1. ***THIS ONE IS LIST OBJECT***

***2) WE NEED TO SORT ACCORDING TO DEFAULT SORTING ORDER BY sort (list l)***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ***-1***  ***A***  ***0*** | ***-2***  ***B***  ***1*** | ***-3***  ***G***  ***2*** | ***-4***  ***K***  ***3*** | ***-5***  ***M***  ***4*** | ***-6***  ***S***  ***5*** | ***-7***  ***Z***  ***6*** |

***Then I need to apply search method which follows binary search***

***\*\*\*\*\*\* Above -1,-2 are insertion point***

***Collections.binarySearch(list l,”z”); // 6***

***Collections.binarySearch(list l,”y”); // -7***

***NOTE: if we are providing our own customize sorting order then then we should pass comparator object otherwise we will get unpredictable answer.***

***Reversing element of list:***

***Public static void reverse (List l)***

***We have one another utility class for Array in util packge***

***Arrays class has several utility methods for array objecs.***

1. ***Sorting element of Array***
2. ***Public static void sort(primitive [ ] p)***

***// sort according to default sorting. We cant do customizer sorting order here.bcz customizer sorting is only for OBJECT NOT FOR PRIMITIVE TYPE.***

1. ***Public static void sort(Object [ ] o)***

***// according to natural sorting order.***

1. ***Public static void sort(Object [ ] o, comparator c )// according to cust. Sorting order.***

***/\* int [ ]a = new int [3]{10,2,5}***

***\*Arrays. sort( a)***

1. ***Searching for array***
2. ***Public static int binarySearch(primitive [ ] p,primitive target)***
3. ***Public static int binarySearch(Object [ ] o, Object target)***
4. ***Public static int binarySearch(Object [ ] o ,Object target, Comparator c)***

***All concepts are same as collections searching.\****

***If we want to convert array into list***

***Public static List asList(Object [ ] o);***

***String [ ] s={ “a”, “b”, “C”};***

***List l=Arrays.asList(s);***

***It will not create the list object . it will just show in the form of list.***

|  |  |  |
| --- | --- | --- |
| ***A*** | ***B*** | ***C*** |

***S***

***L \*\*\*\*\* if we do any change trough any reference variable then change will be performed there.***

***BUT remember we cant increase the size even if we are using formula with the list reference . bcz Array will be dominated over list. so we cant perform any operation which varies the size otherwise we will get RE.***

***And we cant insert hetro object in this type bcz internally it is array object which we defined already.***