|  |  |
| --- | --- |
| Array | Collections |
| Fixed Size | Growable |
| Applies to both primitive and non primitive | Not applicable to primitive types only objects |
| Not based on standard DS so no readymade methods | Based on standard DS and readymade methods are available |
| Good performancewise | Not good performancewise |
| Spacewise bad | Spacewise good |
| Can hold only homogeneous elements | Can hold both homo and heterogeneous elements |

|  |  |
| --- | --- |
| Java | C++ |
| Collection | Container |
| Collection Framework | STL(Standard Library Template) |

9 interfaces in collection framework 🡪

(1)Collection(I) :🡪 it is the root of the collections framework. It defines most common methods which are applicable to any collection objects.

But there is no concrete class that implements collections directly.

|  |  |
| --- | --- |
| Collection | Collections |
| It is an interface | It is a class |
| It is an interface which can be used to represent  A group of individual objects as single entity | It is an utility class which is present in java.util. package to define several methods like searching and sorting for collection objects. |

(2) List(I) :🡪 It is a child interface of Collection.

If we want duplicates are allowed and insertion order is preserved we go for list.

Collection(I) [in java 1.2 v]

List(I) [in java 1.2 v]

Linked List [1.2 V] ArrayList[1.2 V] Vector [1.0 v] legacy

Stack

(3) Set(I) :🡪 here duplicates are not allowed and insertion order is not preserved.

Collection(I) [in java 1.2 v]

Set(I) [in java 1.2 v]

HashSet(1.2)

LinkedHashSet(1.4)

|  |  |
| --- | --- |
| List | Set |
| Duplicates are allowed | Not allowed |
| Insertion order preserved | Not preserved |

(4) SortedSet (I) :🡪 child interface of Set.

When we want to insert non duplicate elements in sorted order then we use this. It also came in 1.2 V

(5)NavigableSet(I) :-> child interface of SortedSet.It defines various methods for navigation purpouse.

It came in 1.6 version. It’s implementation class is TreeSet which came in 1.2 v itself.

Set(I)

SortedSet(I)

NavigableSet(I)

TreeSet

(6)Queue :🡪 It is a child of Queue interface. It came in 1.5v. If we want to store objects before processing we use queue. Queue(I)

PriorityQueue BlockingQueue

LinkedBlockingQueue PriorityBlockingQueue

(7)Map:🡪 Not a child of Collection(I). A group of individual objects as key value pair used in Map.

Both key and values are object. No duplicate key but duplicate values allowed.

Map(I)[1.2] Dictionary(Abstract Class)

HashMap[1.2] WeakHashMap[1.2] IdentityHashMap[1.2] Hashtable[1.0]

LinkedHashMap [1.2] Properties[1.0]

(8) SortedMap :🡪 it is a child interface of map.key value pair according to sorting order in this objects are stored.It came also in 1.2 V.

(9)NavigableMap :🡪 child interface of SortedMap for navigation purpouse. TreeMap is the implementation of it. Map(I)

Sorted Map (I)

Navigable Map (I)

Tree Map

Overview of Collections

Sorting

If we want default sorting order then to use Comparable Interface.

If we want to implement our own sorting logic then Comparator Interface.

Cursors

Enumeration , Iterator and ListIterator are three interfaces to get data one by one from collections like cursors.

Utility Classes

Collections and Arrays are two classes having utility methods to do operations.

Arrays class defines methods on array objects and Collections class defines methods for collection objects.

(1)Collection Interface

It has common methods :🡪

boolean add(Object o)🡪adds o to the collection.

boolean addAll(Collection c)🡪 adds c to the collection.

boolean remove(Object o)🡪removes o from the collection.

boolean removeAll(Collection c)🡪removes c from the collection.

Void clear() 🡪 removes everything from the collection.

boolean retainAll(Collection c)🡪removes everything from the collections except c .

boolean isEmpty()🡪 is the collection empty.

int size()🡪 gives size.

boolean contains(Object o)🡪 if it contains o.

boolean containsAll(Collection c)🡪 if it contains c.

Object[] toArray()🡪 converts collection objects to object array.

Iterator iterator()🡪to access the collection objects one by one.

Collection interface doesn’t have any methods to retrieve objects.

(2)List Interface

Index playes important role in List. We preserve insertion locations by index and duplicates also we differentiate by indexes.

Methods :🡪 1) void add(int index,Object o),2)boolean addAll(int index,Collection col), 3)Object get(int index), 4)Object remove(int index), 5)Object set(int index, Object new)—retuerns old object and replaces the index of the list with new object, 6) int indexOf(Object o)—returns index of first occurrence of o , 7)int lastIndexOf(Object o)—returns index of last occurrence of o, 8)ListIterator listIterator()—gets the iterator for one by one access.

(2)ArrayList Class

Underlined data structure is Resizable Array or Growable Array.

Duplicates are allowed.

Insertion order preserved.

Null insertion possible.

Heterogeneous objects allowed[Except TreeSet and TreeMap everywhere heterogeneous allowed].

a)ArrayList li=new ArrayList() creates one empty ArrayList with default capacity of 10.Once it reaches its max capacity one new will be created with **capacity=(currentCapacity)\*3/2+1.**

b)ArrayList li=new ArrayList(int initialcapacity)

c)ArrayList al=new ArrayList(Collection c). it converts any collection objects to ArrayList.

System.out.println(collectionobject) it will call toString() method on the collection object which will give the output as **[comma separated collection elements]** as toString() is overridden like this.

Collection objects are used to hold and transfer data from one place to another. To support this every collection implements Serializable and Cloneable interface.

ArrayList and Vector classes implements RandomAccess interface so that any random element can be accessed with the same speed. So if your frequent operation is retrieval then ArrayList is the best choice.

RandomAccess interface is in java.util package. It doesnot contain any methods and it is a marker interface.

ArrayList is the best option if frequent operation is retrieval(because it implements RandomAccess interface) and worst option if the deletion/insertion in the middle is the frequent operation(because several shift are required)

|  |  |
| --- | --- |
| ArrayList | Vector |
| every methods are not synchronized | every methods are synchronized |
| Is not thread safe | Thread safe |
| Performance is high | Performance is low |
| Came in 1.2 version so non legacy | Came in 1.0 version so legacy |

\*\*Get Synchronized version of ArrayList :🡪

Non synchronized : ArrayList l1=new ArrayList();

Synchronized: List l=Collections.synchronizedList(l1); //method is public static List synchronizedList(List l1);

Public static Set synchronizedSet(Set s1) 🡪 we can get synchronized version of Set object.

Public static Map synchronized Map (Set s1) 🡪 we can get synchronized version of Map object.

(3)LinkedList Class

1) Underlined data structure is Doubly Linked List,

2) Insertion order is preserved,

3) Duplicates are allowed,

4) Heterogeneous data can be inserted,

5) Null insertion is possible,

6) Linked List implements Serializable and Clonable interfaces but not Random Access interface,

7) Linked list is best choice if frequent operation is insertion or deletion in the middle,

8) it is worst choice if frequent operation is retrieval.

Methods only in Linked List :🡪 void addFirst(Object o) , void addLast(Object o), Object getFirst(), Object getLast(), Object removeFirst(), Object removeLast(). These are mainly used to build Stack and Queue.

LinkedList Constructor :🡪 (a) LinkedList l1=new LinkedList() creates an empty LinkedList.

(b) LinkedList l1= new LinkedList(Collection c) creates Linked List eqv to other collection objects.

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| Best choice if frequent operation is retrieval | Worst in case of retrieval |
| Worst in case of add/delete in the middle | Best in case of add/delete in the middle |
| Underlying DS is resizable/Growable array | Underlying DS is double linked list |
| It implements RandomAccess interface | It does not implement |

(3)Vector Class

ArrayList and Vector both are same except Vector class methods are synchronized i.e thread safe.

Method capacity() gives the no of elements it can accommodate.

Method size() gives currently no of elements.

Vector class constructors :🡪

(a)Vector v=new Vector() creates a vector object of capacity 10. New capacity of the vector will be when it will be full is =2\*Current Capacity.

(b)Vector v=new Vector(int initialcapacity) creates vector object with initialcapacity. Performance of this is better than first one.  
(c ) Vector v=new Vector(int initialcapacity,int incrementalcapacity) 🡪 specifies incremental capacity also.

(d) Vector v=new Vector(Collection c).

(4)Stack Class

It is child of vector class and its LIFO.

One constructor is there Stack s=new Stack() . it creates empty Stack object.

Methods :🡪 a) push(Object o).

b) pop()

c)peek()🡪 returns top without removing.

d)empty()🡪 returns if empty or not.

e) search(Object o)🡪 returns index if found from the top else -1.

(5) Cursors in Java

Cursors are used to get the objects one by one from the Collection. Three cursors are there 🡪

1)Enumeration, 2)Iterator, 3)ListIterator.

**Enumeration** 🡪 it is introduced in 1.0 v(for Legacy)

We can use it to get Objects one by one from the old Collection Objects.

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

Enumeration e=v.elements();

Two methods of Enumeration 🡪 1)public boolean hasMoreElements(), 2)public Object nextElement().

\*\* There are limitations of Enumeration:

1. It is applicable only for legacy classes ,
2. By using Enumeration we can get only read access and we cannot perform remove operation.
3. To overcome these limitations we should go for Iterator.

**Iterator**🡪 Iterator concept we can apply for any collection object.’

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

Iterator itr=c.iterator() it creates Iterator object on collection objects.

Methods🡪 a)public Boolean hasNext(), b)public Object next() , c) public void remove()

\*\* Limitations of Iterator 🡪

1)Enumeration and Iterator is forward only cursor.

2)Iterator can only remove/read the data. It has no replace and addition capability.

**ListIterator**🡪 it is bidirectional cursor i.e we can move to both direction.

by using it we can perform replacement and addition of new objects easily.

We can create ListIterator Object by using listIterator() method on List objects.

Public ListIterator listIterator() is the method syntax.

\*\* ListIterator is the child interface of Iterator. So by default all the methods of Iterator is there in ListIterator.

Extra Methods are 🡪 a)public int nextIndex(), b)public void remove(), c)public void add(Object o) , d)public void set(Object o) , e) public boolean hasPrevious() , f)public void previous() , g)public int previousIndex().

ListIterator can only be used in List objects.not universal cursor.

|  |  |  |  |
| --- | --- | --- | --- |
| properties | Enumeration | Iterator | ListIterator |
| Applicable for | Only Legacy classes | All collection class | Only List classes |
| movement | Single direction | Single direction | bidirection |
| accessibility | Only read | Read and remove | Read,remove,replace and addition |
| How to get it | element() method of Vector class | Iterator() method of Collection(I) | listIterator() method of List(I) |
| Is it legacy | Yes | No (1,2v) | No (1.2v) |

Vector v=new Vector();

Enumeration e=v.element();

Iterator itr=v.iterator();

ListIterator listitr=v.listIterator();

Sop(e.getClass().getName());🡪 it will give java.util.Vector$1 i.e 1 is the anonymous inner class inside Vector class which implements Enumeration interface.

Similarly,

Sop(itr.getClass().getName());🡪java.util.Vector$ltr.

Sop(listitr.getClass().getName());🡪java.util.Vector$ListItr.

(6) Set(I)

Set is the child interface of collection. Duplicates are not allowed here and insertion order is not preserved.

It does not contain any new methods ,so we have to use Collection Interface methods only.

(7) HashSet(I)

\*Underlying DS is Hashtable.

\*duplicates are not allowed ,if we try to insert duplicates no compile/runtime error will occur.add() method return false.

\*insertion order not preserved and insertion is based on hashcode.

\*heterogeneous objects are allowed and null insertion is also possible.

\*implements Serializable and Clonable but not RandomAccess.

\*HashSet is the best choice if frequent operation is search.

**Constructors of HashSet** :🡪

1)HashSet h=new HashSet() creates empty HashSet object with default capacity 16 and **load factor/fillRatio** 0.75.

LoadFactor 0.75 means after filling 75% of the current length a new Hashset object will be created.

2) HashSet h=new HashSet(int intialCapacity) here initialCapacity is given but load factor is 0.75.

3) HashSet h=new HashSet(int intialCapacity,float loadFactor)

4) HashSet h=new HashSet(Collection c)

(7) LinkedHashSet(I)

\*It is the child class of HashSet. It is introduced in 1.4v.

\*It is exactly same as HashSet except following differences🡪

|  |  |
| --- | --- |
| HashSet | LinkedHashSet |
| Underlying DS is HashTable | Underlying DS is HashTable+LinkedList |
| Insertion order not preserved | Insertion order preserved |
| Introduced in 1.2v | Introduced in 1.4v |

LinkedHashSet is the best choice to develop cache based applications where duplicates are not allowed and insertion order must be preserved.

(8) SortedSet(I)

It is a child interface of Set and we go for this when we want that duplicates are not allowed and some sorting order should be there.

Specific methods for SortedSet Interface 🡪

Object first() :=returns first element of Sorted Set,

Object last():=returns last element.

SortedSet headset(Object obj):=returns the sorted set where elements are <obj

SortedSet tailSet(Object obj):= returns the sorted set where elements are >=obj

SortedSet subset(Object obj1,Object obj2):= returns the SortedSet whose objects lies between obj1 and obj2.

Comparator comparator() :=returns Comparator object describing underlying sorting technique.If default sorting order then it returns null.

Default natural sorting order for number is ascending and for strings are alphabetical order.

(8) TreeSet

Underlying DS is Balanced Tree. Heterogeonous objects are not allowed.,If we try to insert then we will get ClassCastException. Null insertion is allowed but only once.

Constructors :🡪1)TreeSet t=new TreeSet() creates empty TreeSet with natural sorting order.

2) TreeSet t=new TreeSet(Comparator c) creates empty TreeSet with customized sorting order.

3)TreeSet t=new TreeSet(Collection c)

4) TreeSet t=new TreeSet(SortedSet c)

\*\*null insertion 🡪1) For empty TreeSet as the first element null insertion is possible.But after that if we want to try insert any other element we will get NullPointerException.

2)For non empty TreeSet if we try to insert null then also we will get NullPointerException because then the TreeSet will expect the type of the new element equal to the type of the already inserted element.

\*\*If we add StringBuffer objects in the Treeset then we will get exception that ClassCastException : StringBuffer objects are not Comparable.

i.e if we depend on the default sorting order then the object must have to be Homogeneous and Comparable(i.e the corresponding Class of the objects implement Comparable interface).

\*\*StringBuffer class doesn’t implement Comparable interface

(8.1) Comparable(I)

It is in java.lang package. Only one method is there 🡪 public int compareTo(Object obj1) [int return type because the return value can be three value which Boolean cannot cover].

\*To call this method we can write🡪 obj1.compareTo(obj2)

Returns a) –ve means obj1 has to come after obj2.[value is not important only sign is important]

b) +ve means obj1 has to come before obj2. [value is not important only sign is important]

c) 0 means obj1 equal obj2.

If either obj1 or obj2 is null we will get NullPointerException.

\*\*\*If we depend on the default natural sorting order internally JVM will call to compareTo() method while inserting into TreeSet. Hence the objects should be Comparable.

TreeSet t=new TreeSet();

t.add(“B”);

t.add(“Z”); //”Z”.compareTo(“B”) returns +ve.

t.add(“A”);// “A”. compareTo(“B”) returns -ve.

sop(t); //[A B Z].

\*\* If few are not satisfied with default sorting order we go for customized sorting by using Comparator. Comparable meant for default natural sorting order whereas Comparator meant for customized sorting order.

(8.2) Comparator(I)

It is present in java.util package. We go for it when we require customized sorting.

It contains two method🡪 a)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 equal obj2,

b)public Boolean equals()🡪 it is a dummy method.

\*\*\*Whenever we are implementing Comparator interface ,compulsory we should provide implementation for compare() method. Implementing equals() method is optional because it is already available in every java class from Object through inheritance.

\*\*\*Use Of Comparator object for customized Sorting

(1)Let’s say we want to descending sorting order for integer🡪

Class MyComparator implements Comparator

{

Public int compare(Object obj1,Object obj2)

{

if((Integer)obj1>(Integer)obj2)

{

Return –1;

}

else if((Integer)obj1<(Integer)obj2)

{

Return +1;

}

Elseif((Integer)obj1==(Integer)obj2)

{

Return 0;

}

}

}

Public static void main(String[] args){

TreeSet t1=new TreeSet(new MyComparator());

t1.add(10); t1.add(13); t1.add(1); t1.add(5);

sop(t1);

}

It will print descending order of values in t1 as defined by the comparator.

In the compare() method if we return +1 always then the records will be inserted according to insertion order. If we return +1 always then the records will be inserted according to reverse of insertion order.

If we return 0 then only first element will be inserted others it will consider as duplicate.

(2)Let’s say we want to reverse alphabatical sorting order for Strings🡪

Class MyComparator implements Comparator

{

Public int compare(Object obj1,Object obj2)

{

return -(obj1.compareTo(obj2)); //Or we can write obj2.compareTo(obj1);

}

}

This will sort reverse alphabetical order.

(3)Let’s say we want to insert in TreeSet in alphabatical sorting order for StringBuffer🡪

Class MyComparator implements Comparator

{

Public int compare(Object obj1,Object obj2)

{

String s1= obj1.toString();

String s2=obj2.toString();

return (s1.compareTo(s2));

}

}

(4)Let’s say we have String and StringBuffer in TreeSet.The sorting order is increasing length if both have same length then alphabetical order🡪

Public int compare(Object obj1,Object obj2)

{

String s1= obj1.toString();

String s2=obj2.toString();

If(s1.length()>s2.length())

{

Return +1;

}

Else if(s1.length()<s2.length())

{

Return -1;

}

Else

{

Return s1.compareTo(s2);

}

}

\*\*If we want to add custom class objects in TreeSet then the class should have Comparing logic by implementing Comparator interface.

(5) One Employee class to be used in Comparable Interface 🡪 this class has to implement Comparable interface and provide implementation for compareTo() method.

Class Employee implements Comparable

{

String name; int eid;

Public int compareTo(Object obj)

{

Int eid1=this.eid;

Employee e=(Employee)obj;

Int eid2=e.eid;

If(eid1<eid2)

{

Return -1;

}

else If(eid1<eid2)

{

Return 1;

}

Else

{

Return 0;

}

}

}

|  |  |
| --- | --- |
| Comparable | Comparator |
| Default natural sorting order | Customized sorting order |
| Present in java.lang package | Present in java.util package |
| It contains one method-> public int compare(Object obj1) | It contains two method-> public int compareTo(Object obj1,Object obj2). |
| All Wrapper Class and String class implements this interface | There are two classes which implements it are->  Collator and RuleBasedCollator. |

**\*\*Comparison table for set implemented classes**

|  |  |  |  |
| --- | --- | --- | --- |
| Property | HashSet | LinkedHashSet | TreeSet |
| Underlying DS | HashTable | HashTable+LinkedList | Balanced Tree |
| Insertion Order | Not preserved | preserved | Not preserved |
| Sorting Order | Not applicable | NA | applicable |
| Heterogeneous Objects | Allowed | Allowed | Not allowed |
| Duplicate objects | Not allowed | Not allowed | Not allowed |
| Null acceptance | Allowed only once | Allowed only once | Allowed only once for empty tree set as the first element |

Q.))Internal Implementation Of Set/HashSet (How Set Ensures Uniqueness) : Core Java Collection Interview Question

In [core java interview questions](http://javahungry.blogspot.com/2013/09/core-java-coding-programming-questions-answers-technical-interview-java.html) , It is common to get bombarded with Collection framework questions . I was interviewed in Goldman Sachs , and there they asked a question where i got dumbstruck . Interviewer asked How do you implement Set in  Java in other words internal working of Hashset or How hashset works in java. That is , how will make sure each and every element is unique without using Set interfaces or Classes that implements Set Interface .  
  
*Read Also*:   [How hash map works in java](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html)  
  
I gave the answer , although qualified the interview round as well , but the answer is far from satisfactory .  
So I came back to  home and do some research . So finally i got the answer and sharing it with you .  
  
  
**Set Implementation Internally in Java**  
Each and every element in the set is unique .  So that there is no duplicate element in set .  
  
So in java if we want to add elements in the set then we write code like this

**public** **class** **JavaHungry** {

**public** **static** **void** **main**(String[] args)

{

// TODO Auto-generated method stub

HashSet<Object> hashset = **new** HashSet<Object>();

hashset.add(**3**);

hashset.add("Java Hungry");

hashset.add("Blogspot");

System.out.println("Set is "+hashset);

}

}

*It will print the result* :       Set is [3, Java Hungry, Blogspot]

Now let add duplicate element in the above code

**public** **class** **JavaHungry** {

**public** **static** **void** **main**(String[] args)

{

HashSet<Object> hashset = **new** HashSet<Object>();

hashset.add(**3**);

hashset.add("Java Hungry");

hashset.add("Blogspot");

hashset.add(**3**); // duplicate elements

hashset.add("Java Hungry"); // duplicate elements

System.out.println("Set is "+hashset);

}

}

*It will print the result* :       Set is [3, Java Hungry, Blogspot]  
  
  
Now , what happens internally when you pass duplicate elements in the  add() method of the Set object , It will return false and do not add to the HashSet , as the element is already present .So far so good .  
  
But the main problem arises that how it returns false . So here is the answer  
  
When you open the HashSet implementation of the add() method in Java Apis that is rt.jar , you will find the following code in it

**public** **class** **HashSet**<E>

**extends** AbstractSet<E>

**implements** Set<E>, Cloneable, java.io.Serializable

{

**private** **transient** HashMap<E,Object> map;

// Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object PRESENT = **new** Object();

**public** **HashSet**() {

*map =* ***new*** *HashMap<>();*

}

// SOME CODE ,i.e Other methods in Hash Set

**public** **boolean** **add**(E e) {

**return** map.put(e, PRESENT)==**null**;

}

// SOME CODE ,i.e Other methods in Hash Set

}

So , we are achieving uniqueness in Set,internally in java  through HashMap . Whenever you create an object of HashSet it will create an object of HashMap as you can see in the italic lines in the above code .  
We already discussed   [How HashMap works internally  in java](http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html) .  
  
As we know in HashMap each key is unique . So what we do in the set is that we pass the argument in the add(Elemene E) that is E as a key in the HashMap . Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy  value that is ( new Object () ) which is referred by Object reference PRESENT .  
  
So , actually when you are adding a line in HashSet like  hashset.add(3)   what java does internally is that it will put that element E here 3 as a key in the HashMap(created during HashSet object creation) and some dummy value that is Object's object is passed as a value to the key .  
  
Now if you see the code of the HashMap put(Key k,Value V) method , you will find something like this  
  
 public V put(K key, V value) {  
//Some code  
}  
  
The main point to notice in above code is that put (key,value) will return  
  
1.  null , if key is unique and added to the map  
2.  Old Value of the key , if key is duplicate  
  
So , in HashSet add() method ,  we check the return value of map.put(key,value) method with null value   
i.e.  
  
   public boolean add(E e) {  
            return map.put(e, PRESENT)==null;  
       }  
  
So , if map.put(key,value) returns null ,then  
map.put(e, PRESENT)==null      will return true and element is added to the HashSet.  
  
  
  
So , if map.put(key,value) returns old value of the key ,then  
map.put(e, PRESENT)==null      will return false and element is  not added to the HashSet .

**Add and Remove method of HashSet**

[?](http://opensourceforgeeks.blogspot.in/2015/02/understanding-how-hashmap-and-hashset.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public boolean add(E e) {      return map.put(e, PRESENT)==null;      }            public boolean remove(Object o) {      return map.remove(o)==PRESENT;      } |

- See more at: http://opensourceforgeeks.blogspot.in/2015/02/understanding-how-hashmap-and-hashset.html#sthash.OIiaYAw2.dp