**CHAPTER 18**

**CollectionFrameWork**

Java Collections framework API is a unified architecture for representing and manipulating collections. The API contains Interfaces, Implementations & Algorithm to help java programmer in everyday programming. Collection framework is the standardized mechanism of grouping of similar or dissimilar of objects into a single object. This single object is known as collection framework object.

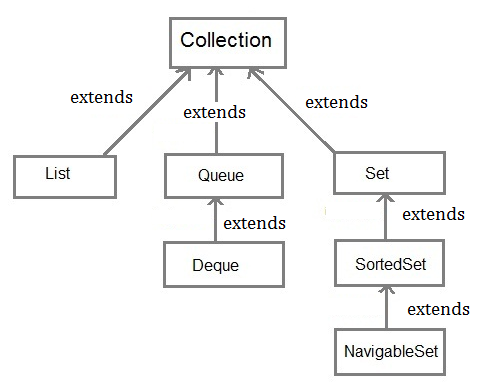
* Reduces effort to learn and to use new APIs.
* Reduces effort to design new APIs.
* Encourages & Fosters software reuse.

To be specific, there are six collection java interfaces. The most basic interface is Collection. Three interfaces extend Collection: Set, List, and SortedSet. The other two collection interfaces, Map and SortedMap, do not extend Collection, as they represent mappings rather than true collections.

**Goals of collection framework:**

1. Collection framework improves the performance of JAVA, J2EE projects. When we want to transfer the bulk amount of data from client to server and server to client, using collection framework we can transfer that entire data at a time).
2. Collection framework allows us to prove similar or dissimilar type of objects
3. Collection framework is dynamic in nature i.e., they are extends (arrays contain the size which is fixed in nature and they allows similar type of data).
4. Collection framework contains adaptability feature (adaptability is process of adding collection object at the end of another collection object).
5. Collection framework is algorithmic oriented (collection framework contains various sorting and searching techniques of data structures as a predefined concepts).
6. In order to deal with collectionframework we must use a package called java.util.\***.**

**Collection Framework Interfaces:**

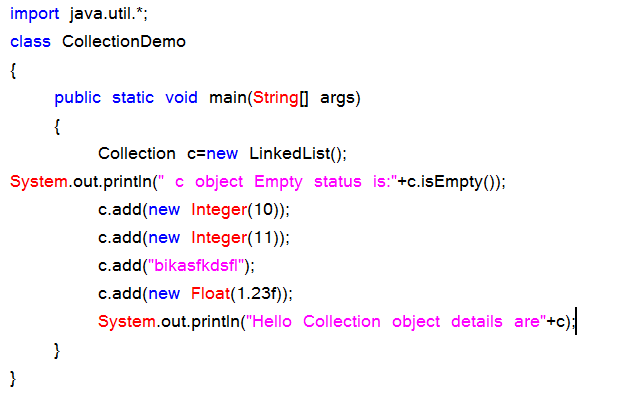


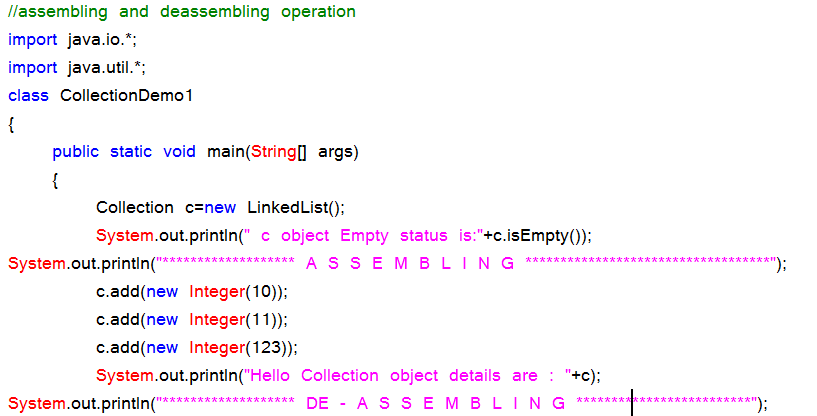
**java.util.Collection:**

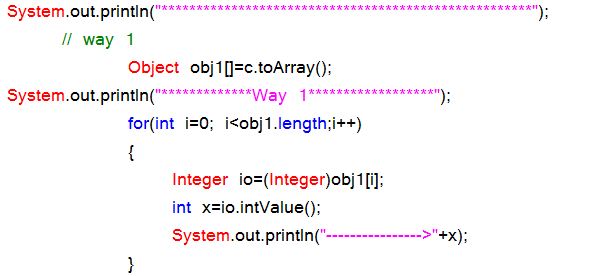
Collection is an interface whose object allows us to organize similar or different type of objects into single object.

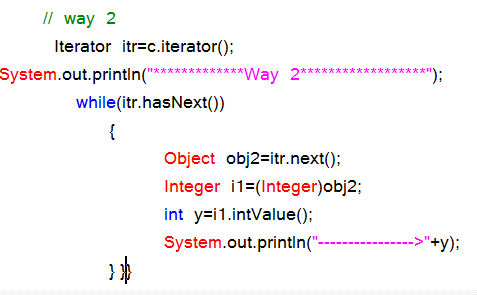
**The Collection interface is having the following features:**

* It is available at the top of the hierarchy of all the interfaces which are available in the collectionframework.
* An object of Collection allows us to add duplicate elements.
* Collection object always displays the data in forward direction only.
* Collection object will print the data on the console in random order.
* Collection object always allows us to insert an element at the end only i.e. we can't insert an element at the specific position.







****

**java.util.List:**

* List is the sub-interface of java.util.Collection
* List object also allows us to add duplicates.
* List object allows us to add an element either at the ending position or at specific position.
* List object allows us to retrieve the data in forward direction, backward direction and random retrieval

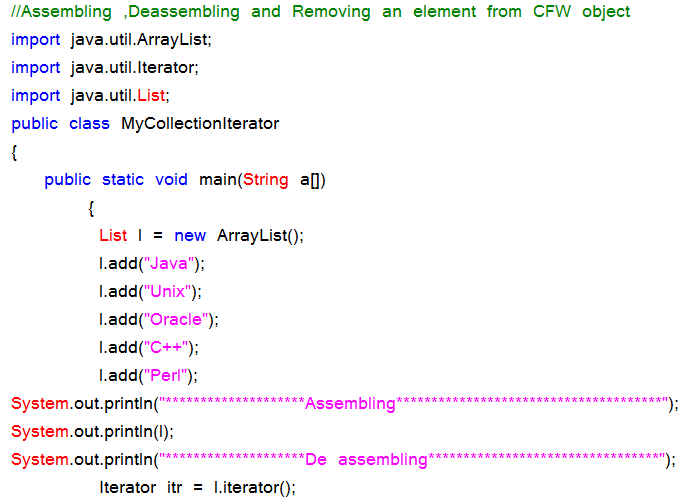
**java.util.Set:**

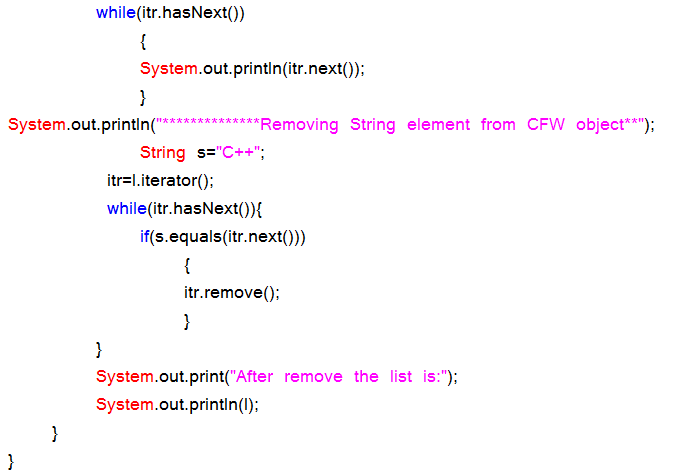
* Set is the sub-interface of java.util.Collection interface.
* An object of Set does not allows duplicates i.e., all the elements in the set must be distinct(unique).
* Set object always displays the data in random order.
* Set object allows us to add the elements only at ending position.
* Set object allows us to retrieve the data only in forward direction.

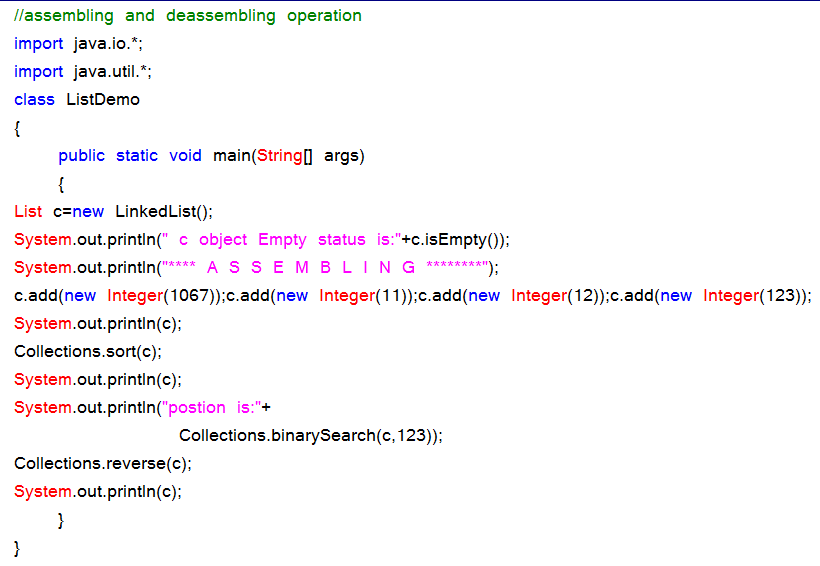
**java.util.SortedSet:**

* SortedSet is the sub-interface of java.util.Set interface.
* SortedSet object does not allow duplicates.
* SortedSet object will displays the data automatically in sorted order.
* SortedSet object allows us to add the elements only at ending position.
* SortedSet object allows us to retrieve only in forward direction.

**Program on List interface:**





****

**Code snippet { ☹ For toArray() }**

Object obj[]=s.toArray ();

for (int i=0; i<obj.length; i++)

{

Integer io= (Integer)obj [i];

int x=io.intValue ();

}

System.out.println(x);

**Iterator interface:**

Iterator itr=c.iterator ();

Int s=0;

While (itr.hasNext ())

{

Object obj=itr.next ();

Integer io= (Integer) obj;

int x=io.intValue ();

}

System.out.println(x);

**Iterator interface:** Iterator is an interface which always uses the extracted data from any Collection object.

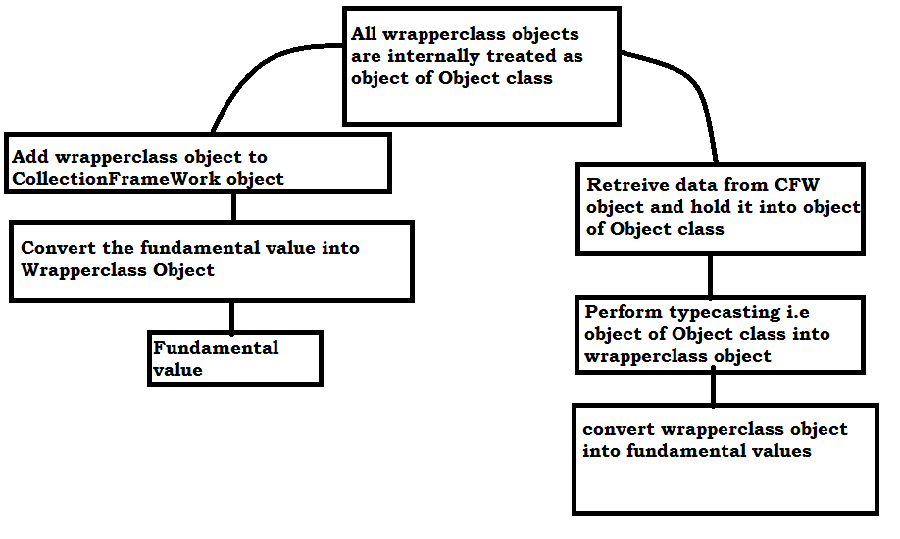
**Methods in Iterator interface:**

1. public boolean hasNext ()

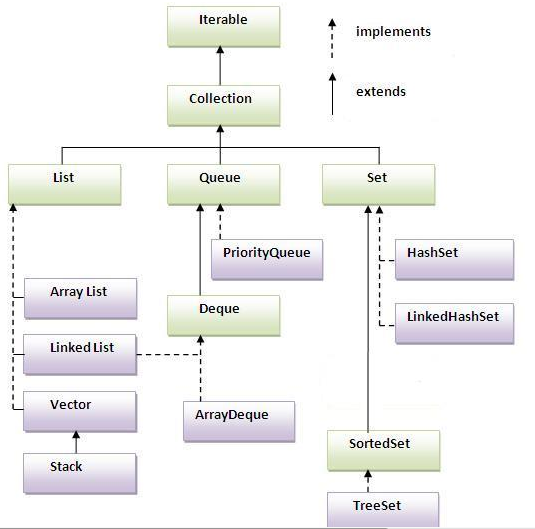
2. public Object next ()

3. public Object remove ()

**1Dimensional CollectionFrameWork:**



**CollectionFrameWork Hierarchy(with classes and interfaces):**



**1D CollectionFrameWork classes:**

* LinkedList
* ArrayList
* HashSet
* TreeSet

**Perform the dataretreival in forward and backward directions:**

import java.util.\*;

public class MyListIterator1

{

public static void main(String args[])

{

LinkedList li = new LinkedList();

li.add(921); li.add(628); li.add(411);

li.add(923); li.add(456);

System.out.println("\*\*\*\*\*\*\*Assembling\*\*\*\*\*\*\*");

System.out.println(li);

System.out.println("\*\*\*\*\*\*\*Assembling at first,last\*\*\*\*\*\*\*");

li.addFirst(1.23f);

li.addLast('M');

li.add("ss");

System.out.println("\*\*\*\*De assembling \*\*\*\*");

ListIterator litr=li.listIterator();

System.out.println("Elements in forward direction");

while(litr.hasNext())

{

System.out.println(litr.next());

}

System.out.println("Elements in backward(Reverse) directiton");

while(litr.hasPrevious())

{

System.out.println(litr.previous());

}

}

}

**Disadvantages of LinkedList:**

1. Additional memory space is created for address part of the node in heap memory.

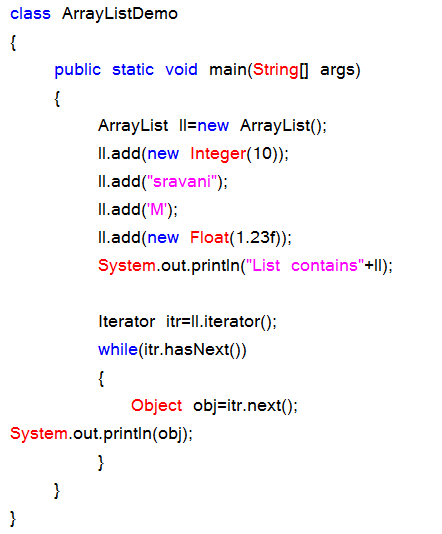
2. Retrieval time is more.

3. Since, we are wasting most of the memory space for addresses, performance will be reduced.

**ArrayList:**

Java ArrayList represents an automatic re-sizable array and used in place of array. Since we can not modify size of an array after creating it, we prefer to use ArrayList in Java which re-size itself automatically once it gets full. ArrayList in Java implements List interface and allow null. Java ArrayList also maintains insertion order of elements and allows duplicates opposite to any Set implementation which doesn't allow duplicates. ArrayList supports both Iterator and ListIterator for iteration but it’s recommended to use ListIterator as it allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the Iterator's current position in the list.

**Program:**



**Advantages of ArrayList over LinkedList:**

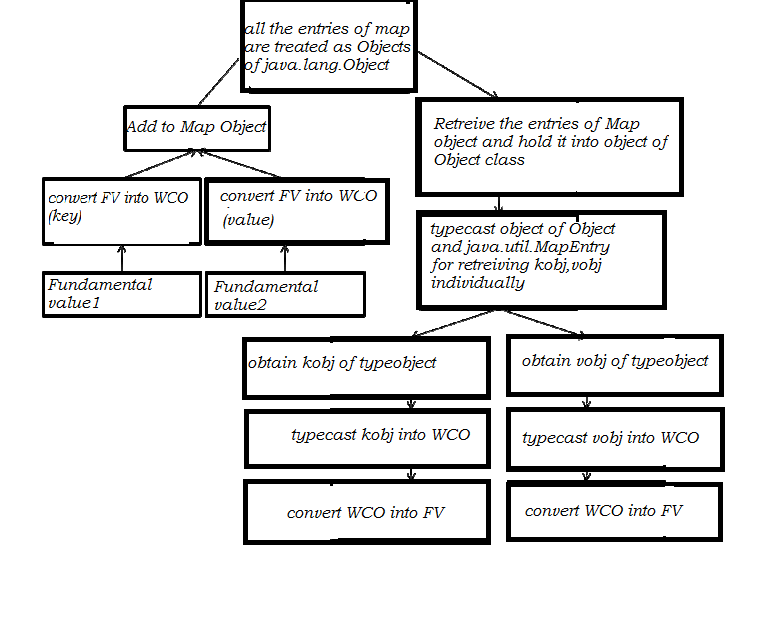
1. No additional memory space is required for data of ArrayList.

2. Retrieval time is quite faster.

3. Performance is high. Since, there is no memory space is required for maintaining address of data of ArrayList.

**Two dimensional framework (or) maps:**

Two dimensional framework organize the data in the form of (key,value) pair. The value of key is an object and they must be unique. The value of value is also an object which may or may not be unique. Two dimensional framework contains collection of interfaces and collection of classes which are also known as map interfaces and map classes.It is very important for other technologies like ABAP, Selenium ,HADOOP.



**java.util.Map:**

Java.util.Map extends Collection. An object of Map allows to organize the data in the form of (key, value) pair. Here key and value must be objects. An object of Map allows displaying the data in that order in which order we have added the data.

**Methods:**

**public boolean put(Object kobj, Object vobj):**

This method is used for adding the data in the form of (key, value). This method returns false. When we are trying to add duplicate key and values. This method returns true as long as we enter unique key objects.

**public boolean putAll (Map):** This method is used for adding one Map object at the end of another Map object.

**public Set entrySet ():** This method is used for obtaining the data of the Map object at end of another Map object.

**public Object get (Object vobj):** This method is used for obtaining value of value by passing value of key object.

**public void remove (Object kobj):** This method is used for removing the entire map entry by passing the value of key object.

**java.util.Map.Entry:**

Here Map is an interface and Entry iis the class in Map interface. java.util.Map.Entry is used for retrieving the data separately in the form of key object and value object from the Map object.obtaining the data of the map in the form of Set

**Methods:**

public Object getKey();

public Object getValue();

Above methods are used for obtaining Keyobject and value object

**Map classes:**

Map classes contain all the definitions for the abstract methods of Map interface. In java.util.\* package we have the following Map classes and whose hierarchy is given below:

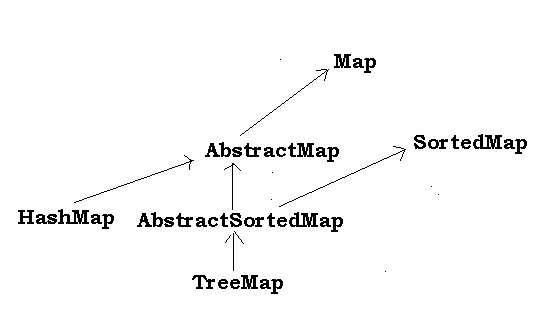
1. AbstractMap implements Map

2. AbstractSortedMap extends AbstractMap implements SortedMap

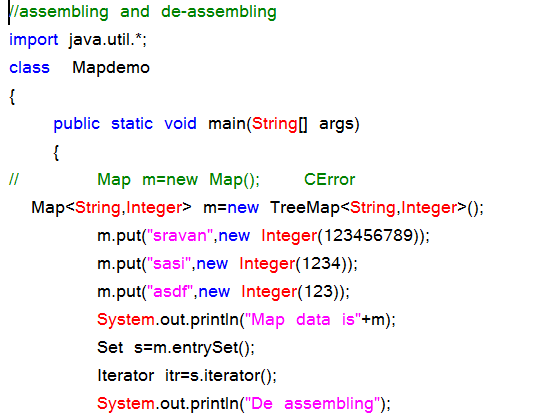
3. HashMap extends AbstractMap

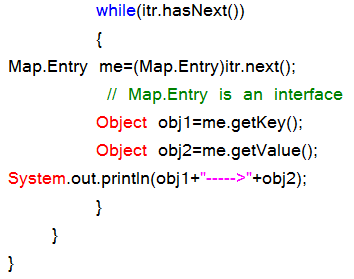
4. TreeMap extends AbstractSortedMap

**Map Hierarchy:**

****

**Program:**

****

****

**HashMap:**

HashMap is a Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key.

The HashMap class is roughly equivalent to Hashtable,except that it is unsynchronized and permits nulls. This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. This implementation provides constant-time performance for the basic operations (get and put).

**Code snippet:**

Set s=hm.entrySet();

System.out.println("HashMap elements after deassembling--------->:");

Iterator itr=s.iterator();

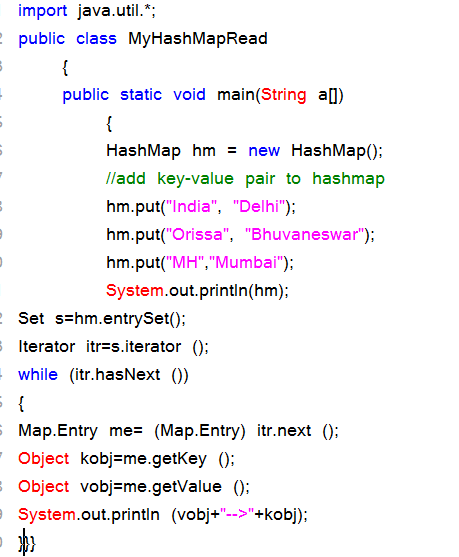
while(itr.hasNext())

{

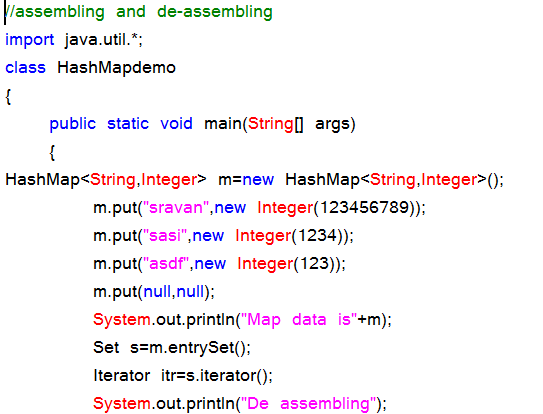
Map.Entry me=(Map.Entry)itr.next();

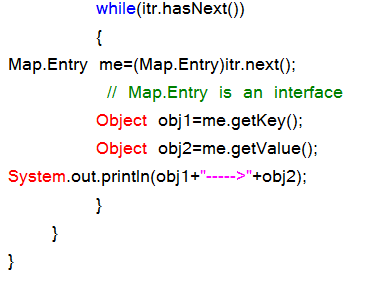
}

**HashMap program:**

****

**Program on HashMap:**

****

****

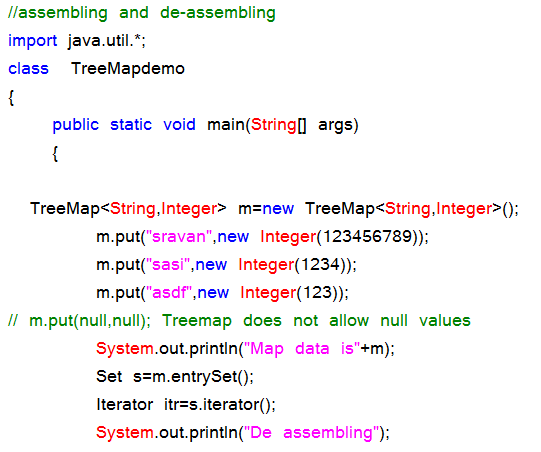
**TreeMap:**

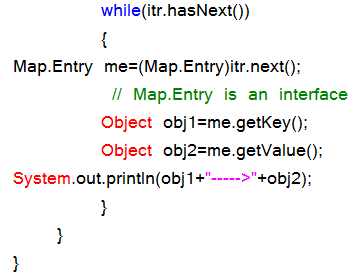
1.TreeMap extends AbstractMap and implements NavigableMap interface

2.It creates Map ,stored in a tree structure

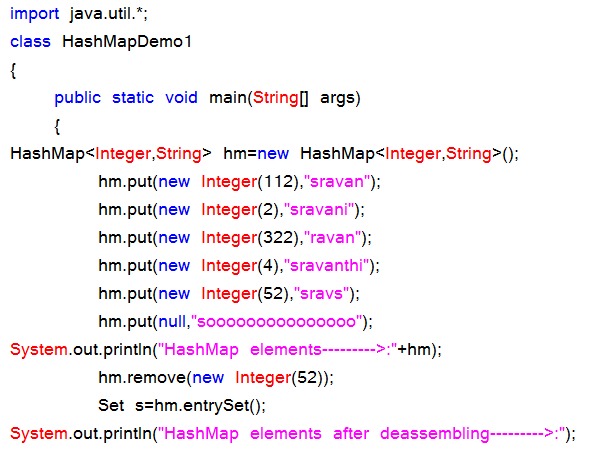
3. A TreeMap provides an efficient meaning of storing key/value pair in effecient order

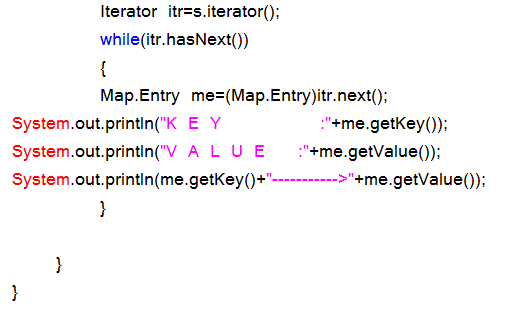
4.It provides key/value pairs in sorted order and allows rapid retrieval.

****

****

**HashMap Demo:**

****

****

**Differences HashSet and TreeSet:**

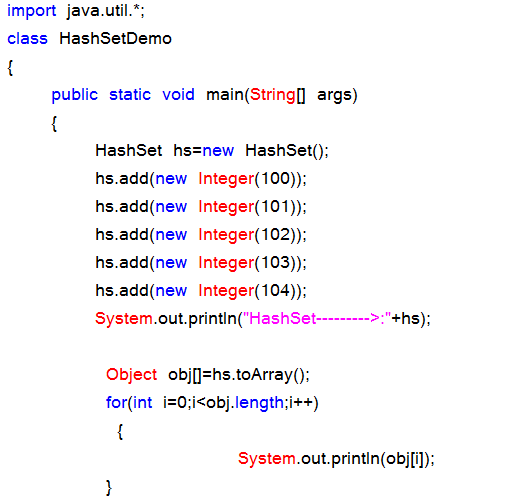
**Treeset does not allow heterogenous elements, null insertion is also not possible**

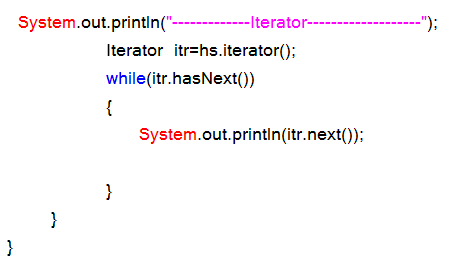
|  |  |
| --- | --- |
| **HashSet** | **TreeSet** |
| 1. It extends AbstractSet. | 1. It extends AbstractSet and implements SortedSet. |
| 2. It follows hashing mechanism to organize its data. | 2. It follows binary trees (AVL trees) to  organize the data. |
| 3. We cannot determine in which order it displays its data. | 3. It always displays the data in sorted  order. |
| 4. Retrieval time is more. | 4. Retrieval time is less. |
| 5. The operations like insertion, deletion and modifications takes more amount of time. | 5. The operations like insertion, deletion  and modifications take very less time. |
| 6. Creating HashSet is nothing but creating an object of HashSet () class. | 6. Creating TreeSet is nothing but  creating an object of TreeSet () class. |

**HashMap and TreeMap differences:**

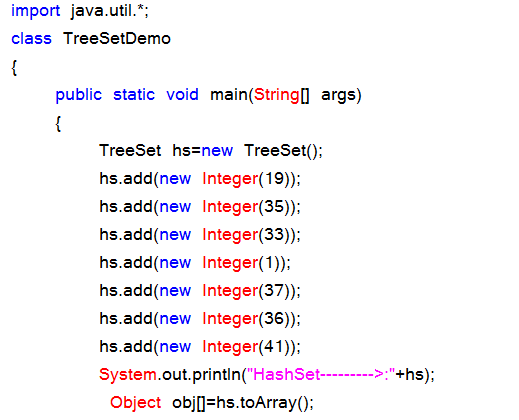
|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1. It extends AbstractMap. | 1. It extends AbstractSortedMap |
| 2. It follows hashing mechanism. | 2. It follows binary tree concept to obtain  data in (k, v) form. |
| 3. Retrieval time is more. | 3. Retrieval time is less. |
| 4. Insert, update and delete operations  takes more time. | 4. Insert, update and delete operations  takes less time. |
| 5. Creating an object of HashMap | 5. Creating an object of TreeMap. |
| 6. Random order. | 6. Sorted order. |
| 7.HashMap is can contain only one null key | 7.TreeMap cant contain any null key |

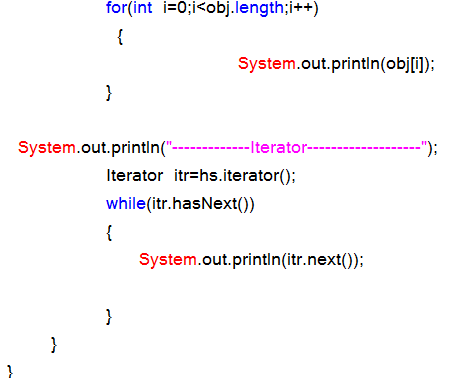
**HashSet Demo:**

****

****

**TreeSet Demo:**

****

****

**Creating Hashtable is nothing but creating an object of Hashtable class.(it is a legacy CFW class)**

**Legacy(old) collection Framework:**

When SUN Micro Systems has developed java, collection framework was known as data structures. Data structures in java were unable to meet industry requirements at that time. Hence data structures of java was reengineered and they have added ‘n’ number of classes and interfaces and in later stages the data structures of java is known as new collection framework.

**Interfaces:**

We have only one interface, namely **java.util.Enumeration**. This interface is used for extracting the data from legacy collection framework classes

**Classes:**

As a part of legacy collection framework we have the following essential classes: **Vector, Stack, Dictionary, Hashtable and properties**. Here, Vector and Stack belongs to one dimensional classes whereas Dictionary, Hashtable and Properties belongs to two dimensional classes.

* **What is the difference between normal collection framework and legacy collection framework?**

**Answer:** All the classes in the normal collection framework are by default belongs to non synchronized classes whereas all classes in legacy collection framework are by default belongs to synchronized classes

**Vector:**

Its functionality is exactly similar to ArrayList but Vector class belongs to synchronized,whereas ArrayList belongs to non-synchronized class.

Creating a Vector is nothing but creating an object of java.util.Vector class.

Null insertion is possible here

**Vector API:**

***Constructors:***

1.Vector (); 2.Vector (int size);

**Instance methods:**

public int size ();

public void addItem (Object obj); [old]

public void addElement (Object obj); [new]

public void addItem (int pos, Object obj);

public void addElement (int pos, Object obj);

public Object getItem (int pos);

public Object remove (int pos);

public void remove (Object obj );

public void removeAll ();

public Enumeration elements ();

**code snippet:**

Enumeration e= v.elements();

While (en.hasMoreElements ())

{

Object obj=en.nextElement ();

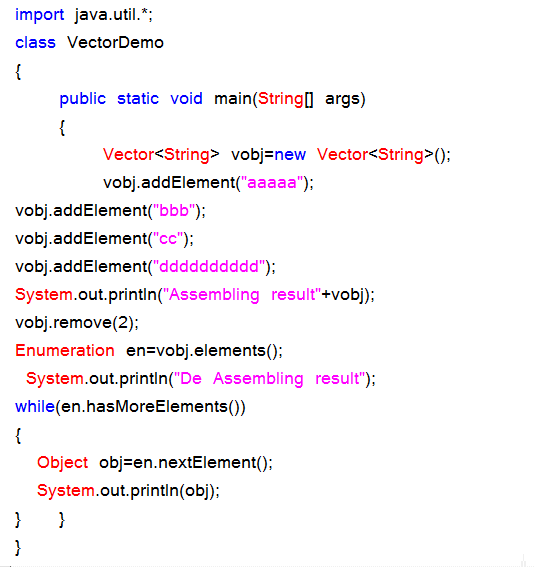
System.out.println (obj);

}

**Methods in Enumeration method:**

public boolean hasMoreElements()

public Object nextElement()

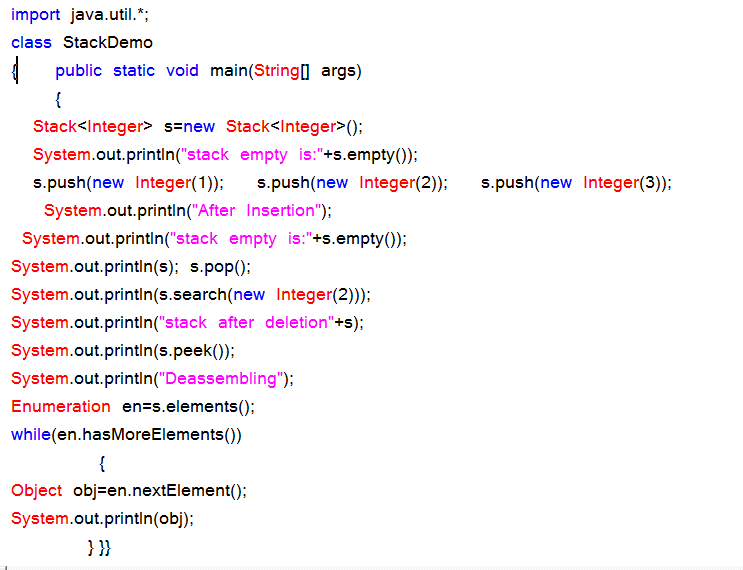


**Stack:** Stack is the sub-class of Vector class. The basic working principal of Stack is Last In First Out.

**Stack API:**

**Constructors:**

1.Stack (); 2.Stack (int size);

****

**Instance methods:**

public boolean empty ();

public void push (Object);

public Object pop ();

public Object peek ();

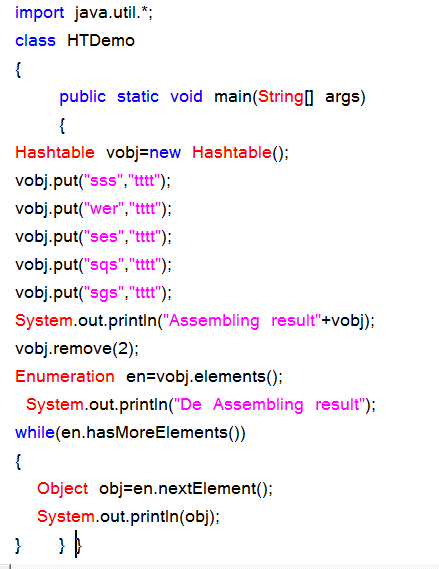
public int search(Object);

**Dictionary:**

Dictionary is an abstract class, whose object allows to retrieve to store the data in the form of (key, value) pair. An object of Dictionary never allows duplicate values as key objects and null values.

**Hashtable:**

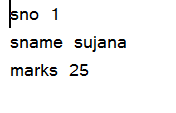
Hashtable is the concrete sub-class of Dictionary and where object allows us to store in the form of (key, value) pair. Hashtable object organizes its data by following hashing mechanism. We cannot determine in which order the Hashtable displays its data.



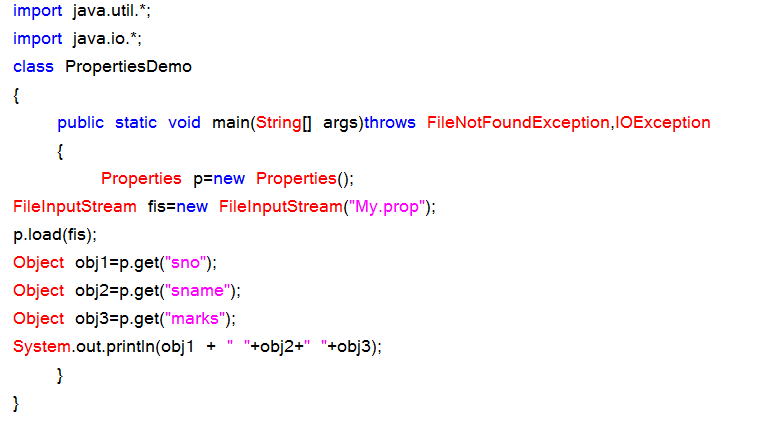
**Properties class:**

Properties is the sub-class of Hashtable class. Properties class object is used for reading of maintaining system environmental variables and reading the data from resource data file or properties file.

**Create a Properties file with My.prop or My.rbf:**



**Properties program:**



**Properties API:**

**Constructor:**

Properties ();

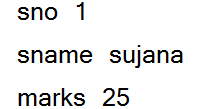
**Instance Methods:**

public void setProperty (Object kobj, Object vobj);

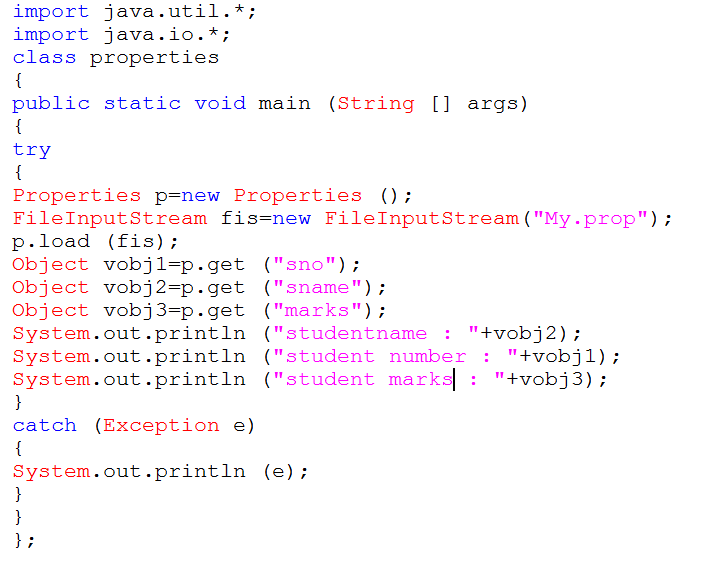
public Object getProperty (Object kobj);

public void load (InputStream);

**Preparation of ResourceBundleFile / Properties file:**



Program on properties file:



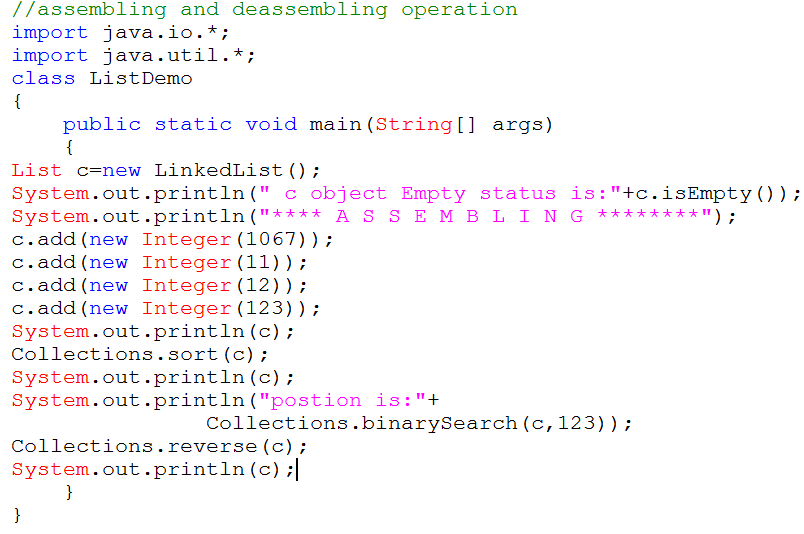
**Collections Class:**

Collections class is a utility class having static methods for doing operations on objects of classes which implement the Collection interface. For example, Collections has methods for finding the max element in a Collection.

Collections is merely an utility method class for doing certain operations, for example adding thread safety to your ArrayList instance by doing this:

List list = Collections.synchronizedList(new Arraylist());

**Source code:**



**Comparable interface** is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object).It provide only single sorting sequence i.e. you can sort the elements on based on single datamember only.For instance it may be either rollno,name,age or anything else.

**Syntax:**

public int compareTo(Object obj): is used to compare the current object with the specified object.

We can sort the elements of:

* String objects
* Wrapper class objects
* User-defined class objects

This interface present in java.lang package & contains only one method that is "compareTo()".

**Method:** public int compareTo(Object obj)

obj1.compareTo(Obj2 ) 🡪 1

1) if it returns -ve value obj1 has to come before obj2

2)if it returns +ve value obj1 has to come after obj2

3)if it returns 0 value obj1 , obj2 are equal

**Example:**

import java.util.\*;

class Test

{

public static void main(String args[])

{

System.out.println("A".compareTo("Z"));

System.out.println("Z".compareTo("K"));

System.out.println("A".compareTo("A"));

}

}

**Ex:**

import java.util.\*;

class Test1

{

public static void main(String args[])

{

TreeSet t=new TreeSet();

t.add("Z");

t.add("K");

t.add("D");

t.add("M");

t.add("D");

System.out.println(t); //**[ D , K , M , Z ]**

}

}

**Program:**

class Student implements Comparable{

int rollno;

String name;

int age;

Student(int rollno,String name,int age){

this.rollno=rollno;

this.name=name;

this.age=age;

}

public int compareTo(Object obj){

Student st=(Student)obj;

if(age==st.age)

return 0;

else if(age>st.age)

return 1;

else

return -1;

}

}

**Simple.java:-**

import java.util.\*;

import java.io.\*;

class Simple

{

public static void main(String args[])

{

ArrayList al=new ArrayList();

al.add(new Student(103,"Adithya",27));

al.add(new Student(101,"Ram",22));

al.add(new Student(102,"Shyam",29));

Collections.sort(al);

Iterator itr=al.iterator();

while(itr.hasNext()){

Student st=(Student)itr.next();

System.out.println(st.rollno+""+st.name+""+st.age);

}

}

}

**Public Comparator comparator():-**

It returns Comparator object describes underlying Sorting technique.if we used default natural sorting order then we will get null.

**Note:**

1) The default natural sorting order for the numbers is ascending order

2) The default natural sorting order for characters & strings is

alphabetical order(Dictionary based order)

**Note:** If we are not satisfied with default natural sorting order.Then we can define

our own customized sorting by using comparator.

* Comparable meant for default Natural sorting order
* Comparator meant for customized sorting order

Collections class provides static methods for sorting the elements of collection. If collection elements are of Set type, we can use TreeSet. But We cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

**Method of Collections class for sorting List elements**

public void sort(List list): is used to sort the elements of List.List elements must be of Comparable type.

**Note:** String class and Wrapper classes implements the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

Example of Sorting the elements of List that contains user-defined class objects on age basis

**public void sort(List list,Comparator c):** is used to sort the elements of List by the given comparator.

Example of sorting the elements of List that contains user-defined class objects on the basis of age and name.In this example, we have created 4 java classes:

AgeComparator.java,NameComparator.java,Simple.java,Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

**Student.java**

class Student{

int rollno;

String name;

int age;

Student(int rollno,String name,int age){

this.rollno=rollno;

this.name=name;

this.age=age;

}}

**AgeComparator.java:**

This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2 , 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

**Program:**

import java.util.\*;

class AgeComparator implements Comparator{

public int Compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

if(s1.age==s2.age)

return 0;

else if(s1.age>s2.age)

return 1;

else

return -1;

}

}

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

**Program:**

import java.util.\*;

class NameComparator implements Comparator{

public int Compare(Object o1,Object o2){

Student s1=(Student)o1;

Student s2=(Student)o2;

return s1.name.compareTo(s2.name);

}

}

**Simple.java:**In this class, we are printing the objects values by sorting on the basis of name and age.

import java.util.\*;

import java.io.\*;

class Simple{

public static void main(String args[]){

ArrayList al=new ArrayList();

al.add(new Student(101,"Vijay",23));

al.add(new Student(106,"Ajay",27));

al.add(new Student(105,"Jai",21));

System.out.println("Sorting by Name...");

Collections.sort(al,new NameComparator());

Iterator itr=al.iterator();

while(itr.hasNext()){

Student st=(Student)itr.next();

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

System.out.println("sorting by age...");

Collections.sort(al,new AgeComparator());

Iterator itr2=al.iterator();

while(itr2.hasNext()){

Student st=(Student)itr2.next();

System.out.println(st.rollno+" "+st.name+" "+st.age);

}

}

}

**Output**: Sorting by Name...

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age...

105 Jai 21

101 Vijay 23

106 Ajay 27

**Collection Interface methods:**

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **boolean add(Object obj)** Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates. |
| 2 | **boolean addAll(Collection c)** Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false. |
| 3 | **void clear( )** Removes all elements from the invoking collection. |
| 4 | **boolean contains(Object obj)** Returns true if obj is an element of the invoking collection. Otherwise, returns false. |
| 5 | **boolean containsAll(Collection c)** Returns true if the invoking collection contains all elements of c. Otherwise, returns false. |
| 6 | **boolean equals(Object obj)** Returns true if the invoking collection and obj are equal. Otherwise, returns false. |
| 7 | **int hashCode( )** Returns the hash code for the invoking collection. |
| 8 | **boolean isEmpty( )** Returns true if the invoking collection is empty. Otherwise, returns false. |
| 9 | **Iterator iterator( )** Returns an iterator for the invoking collection. |
| 10 | **boolean remove(Object obj)** Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false. |
| 11 | **boolean removeAll(Collection c)** Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false. |
| 12 | **boolean retainAll(Collection c)** Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false |
| 13 | **int size( )** Returns the number of elements held in the invoking collection. |
| 14 | **Object[ ] toArray( )** Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements. |
| 15 | **Object[ ] toArray(Object array[ ])** Returns an array containing only those collection elements whose type matches that of array. |

**Assembling and De-assembling with collection interface?**

**Program: (Assembling and de assembling on collection)**

import java.util.\*;

class Collectiondemo

{

public static void main(String[] args)

{

Collection c=new ArrayList();

System.out.println("Initially collection data c is :"+c);

System.out.println("Size of the collection object is :"+c.size());

System.out.println("is collection object Size isEmpty(true/false) :"+c.size());

//adding object data to Collection

c.add(new Integer(10));

c.add(new Integer(11));

c.add(new Integer(12));

c.add(new Integer(13));

System.out.println("is collection object Size isEmpty(true/false) :"+c.size());

System.out.println("After adding Size of the collection object is :"+c.size());

System.out.println("collection data is :"+c);

//checking the object is available (or) not (simply searching)

System.out.println("collection data contains"+new Integer(12)+" is "+c.contains(new Integer(12)));

System.out.println("collection data is :"+c);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

**//creating of another collection object**

Collection c1=new LinkedList();

System.out.println("Initially collection data c1 is :"+c1);

c1.add(new Integer(100));

c1.add(new Integer(101));

c1.add(new Integer(102));

c1.add(new Integer(103));

System.out.println("after adding data to collection c1 is :"+c1);

System.out.println("Size of the collection object is :"+c1.size());

**// boolean add(Object obj)**

System.out.println("adding one object to another object is :"+c.addAll(c1));

System.out.println(" collection data c after adding c1 is :"+c);

**// int hashCode( )** --> Returns the hash code for the invoking collection

System.out.println("Hash code of c is :"+c.hashCode());

System.out.println("Hash code of c1 is :"+c1.hashCode());

**//check objects are equal or not**

System.out.println("Both objects are equal is "+c.equals(c1));

**//observe c,c1 object data**

System.out.println("collection data c is :"+c);

System.out.println("collection data c1 is :"+c1);

**// retainAll🡪 removes data from c object i.e except c1**

System.out.println("Both objects are equal is "+c.retainAll(c1));

System.out.println("collection data c1 is :"+c1);

System.out.println("collection data c is :"+c);

**//After performing De-assembling the object data as fallows(way 1)**

Iterator itr=c.iterator();

System.out.println("After Deassembling Data as Fallows");

while(itr.hasNext())

{

System.out.println(itr.next()); }

**//After performing De-assembling the object data as fallows(way 2)**

Object obj[]=c.toArray();

System.out.println("After Deassembling Data as Fallows");

for(int i=0;i<obj.length;i++)

{

**//object type conversion(type cast) (object data to Integer data)**

Integer io=(Integer)obj[i];

**//apply xxxValue() to fundamental data (WCO--> PDT)**

int value=io.intValue();

System.out.println(io);

} } }

**List Interface:**

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **void add(int index, Object obj)** Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. |
| 2 | **boolean addAll(int index, Collection c)** Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise. |
| 3 | **Object get(int index)** Returns the object stored at the specified index within the invoking collection. |
| 4 | **int indexOf(Object obj)** Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned. |
| 5 | **int lastIndexOf(Object obj)** Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned. |
| 6 | **ListIterator listIterator( )** Returns an iterator to the start of the invoking list. |
| 7 | **ListIterator listIterator(int index)** Returns an iterator to the invoking list that begins at the specified index. |
| 8 | **Object remove(int index)** Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one |
| 9 | **Object set(int index, Object obj)** Assigns obj to the location specified by index within the invoking list. |
| 10 | **List subList(int start, int end)** Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object. |

**Program on List Interface:**

import java.util.\*;

class Listdemo

{

public static void main(String[] args)

{

List l=new ArrayList();

l.add("Nag");

l.add("kumar");

l.add("Mtech");

l.add("hyd");

l.add("kumar");

System.out.println("List data is"+l);

//De assembling - way 1

**System.out.println("Data retreival in Fwd direction");**

ListIterator litr=l.listIterator();

while(litr.hasNext())

{

Object obj=litr.next();

String s=(String)obj;

System.out.println(s);

}

**System.out.println("Data retreival in Backward direction");**

while(litr.hasPrevious())

{

Object obj1=litr.previous();

String s1=(String)obj1;

System.out.println(s1);

}

**//De assembling - way 2**

System.out.println("Data retreival in Fwd direction");

Object obj[]=l.toArray();

for(int i=0;i<obj.length;i++)

{

System.out.println((String)obj[i]);

}

}

}

**Set Interface:**

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **add( )** Adds an object to the collection |
| 2 | **clear( )** Removes all objects from the collection |
| 3 | **contains( )** Returns true if a specified object is an element within the collection |
| 4 | **isEmpty( )** Returns true if the collection has no elements |
| 5 | **iterator( )** Returns an Iterator object for the collection which may be used to retrieve an object |
| 6 | **remove( )** Removes a specified object from the collection |
| 7 | **size( )** Returns the number of elements in the collection |

**Program on Set interface:**

import java.util.\*;

class Setdemo

{

public static void main(String[] args)

{

Set s=new HashSet();

//Perform Assembling

s.add("Nag");

s.add("kumar");

s.add("Mtech");

s.add("hyd");

s.add("kumar");

System.out.println("set data s is :"+s);

//Perform De-assembling

System.out.println("After Deassembling data is");

Iterator itr=s.iterator();

while(itr.hasNext())

{

System.out.println(itr.next());

}

}

}

**SortedSet:**

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **Comparator comparator( )** Returns the invoking sorted set's comparator. If the natural ordering is used for this set, null is returned. |
| 2 | **Object first( )** Returns the first element in the invoking sorted set. |
| 3 | **SortedSet headSet(Object end)** Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set. |
| 4 | **Object last( )** Returns the last element in the invoking sorted set. |
| 5 | **SortedSet subSet(Object start, Object end)** Returns a SortedSet that includes those elements between start and end.1. Elements in the returned collection are also referenced by the invoking object. |
| 6 | **SortedSet tailSet(Object start)** Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object. |

**Source code:**

import java.util.\*;

class SortedDemo

{

public static void main(String[] args)

{

SortedSet s=new TreeSet();

//FV--->WCO

Integer i=new Integer(123);

Integer j=new Integer(1);

Integer k=new Integer(112);

Integer l=new Integer(1121);

Integer m=new Integer(11211);

Integer n=new Integer(99999);

//Perform Assembling

s.add(i);

s.add(j);

s.add(k);

s.add(l);

s.add(m);

s.add(n);

System.out.println("Sorted set data s is :"+s);

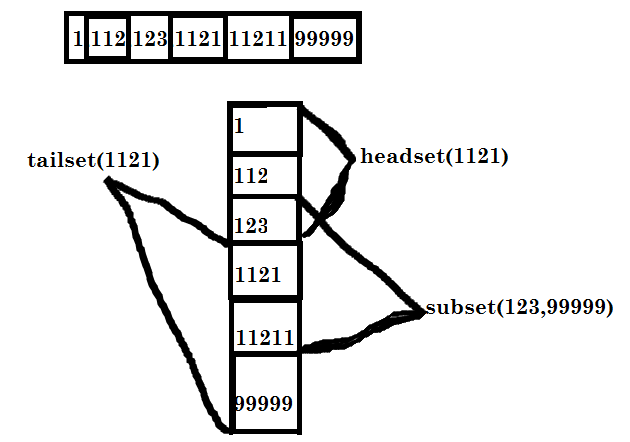
System.out.println("head set data is"+s.headSet(l));

System.out.println("Tail set set data is"+s.tailSet(l));

System.out.println("sub set data is"+s.subSet(i, n));

}}

**Diagramatic Representation:**



**Queue:**

import java.util.LinkedList;

import java.util.Queue;

public class Queuedemo

{

public static void main(String[] args)

{

Queue q=new LinkedList();

q.add(new Integer(1));

q.add(new Integer(2));

q.add(new Integer(33));

q.add(new Integer(4));

q.add(new Integer(12));

q.add(new Integer(11));

System.out.println("Queue Contains--->"+q);

System.out.println(q.peek());

System.out.println("Remove an element from the queue is "+q.remove());

System.out.println("after removal Queue Contains--->"+q);

System.out.println("Queue contains 24 is "+q.contains(24));

System.out.println("Queue contains 12 is "+q.contains(12));

System.out.println("Remove an element from the queue is "+q.remove(12));

System.out.println("Queue contains 12 is "+q.contains(12));

}

}

**ArrayList:**

import java.util.\*;

public class ArrayListdemo

{

public static void main(String[] args) {

ArrayList al=new ArrayList();

al.add("A");

al.add("B");

al.add("C");

al.add("D");

al.add("E");

al.add("F");

System.out.println("ArrayList Elements are"+al);

//assign capacity

al.ensureCapacity(7);

al.add("G");

al.add("H");

al.add("I");

al.add("J");

System.out.println("ArrayList Elements are"+al);

System.out.println("After De assembling");

Object obj[]=al.toArray();

for(int i = 0;i<obj.length;i++)

{

System.out.println((String)obj[i]);

}

}}

**List adds data at any position:**

import java.util.\*;

class ListDemo

{

public static void main(String[] args)

{

List l=new LinkedList();

l.add(new Integer(123));

l.add(new Integer(125));

System.out.println("Assembling data is"+l);

ListIterator itr=l.listIterator();

System.out.println("De assembling data FWD");

while(itr.hasNext())

{

System.out.println(itr.next());

}

System.out.println("De assembling data FWD");

while(itr.hasPrevious())

{

System.out.println(itr.previous());

}

l.add(1,888);

System.out.println("Assembling data is"+l);

}

}

**LinkedList:**

**import** java.util.\*;

**public** **class** LinkedListdemo

{

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

{

LinkedList ll=**new** LinkedList();

ll.add("A");

ll.add("simple");

ll.add("A");

ll.add("Hyderabad");

System.out.println("Assembling data is"+ll);

ll.addFirst("MY");

System.out.println("after adding data at first Assembling data is"+ll);

ll.addLast("Sample");

System.out.println("after adding data at Last Assembling data is"+ll);

ll.push("ttt");

System.out.println("peek first "+ll.peekFirst());

System.out.println("peek last "+ll.peekLast());

System.out.println("poll first "+ll.pollFirst());

System.out.println("poll last "+ll.pollLast());

//De assembling data (way 1)

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*DE ASSEMBLING(way 1)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Iterator itr=ll.iterator();

**while**(itr.hasNext())

{

System.out.println(itr.next());

}

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*DE ASSEMBLING(way 2)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Object obj[]=ll.toArray();

**int** i;

**for**(i=0;i<obj.length;i++)

{

System.out.println(obj[i].toString());

}

}

}

**Categorisation of data with deassembling operation:**

**import** java.util.\*;

**public** **class** ArrayListdemo

{

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

{

ArrayList al=**new** ArrayList();

al.add(**new** Integer(1234));

al.add(**new** Integer(1245));

al.add(**new** Float(123.45f));

al.add(**new** Double(1.235d));

al.add(**new** Character('M'));

al.add(**new** Boolean(**true**));

al.add(**new** String("Nag"));

al.add(**new** Date());

System.*out*.println("ArrayList data is--------->"+al);

/\*De assembling operation way 1\*/

Object obj[]=al.toArray();

**int** i;

**for**(i=0;i<obj.length;i++)

{

System.*out*.println(obj[i]);

}

System.*out*.println("Categorisation of Data as Fallows");

**for**(i=0;i<obj.length;i++)

{

Object objs=obj[i];

**if**(objs **instanceof** Integer)

{

System.*out*.println("Integer data is : "+objs);

}

**else** **if**(objs **instanceof** String)

{

System.*out*.println("String data is : "+objs);

}

**else** **if**(objs **instanceof** Float)

{

System.*out*.println("float data is : "+objs);

}

**else** **if**(objs **instanceof** Character)

{

System.*out*.println("character data is : "+objs);

}

**else** **if**(objs **instanceof** Boolean)

{

System.*out*.println("Boolean data is : "+objs);

}

**else** **if**(objs **instanceof** Double)

{

System.*out*.println("double data is : "+objs);

}

**else** **if**(objs **instanceof** Date)

{

System.*out*.println("Today date is : "+objs);

}

}

}

}

**Example on Map:-**

**import** java.util.\*;

**public** **class** Mapdemo

{

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

{

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

System.*out*.println("Assembling the data");

m.put(**new** Integer(123),**new** String("Anil"));

m.put(**new** Integer(124),**new** String("balu"));

m.put(**new** Integer(122),**new** String("chandu"));

m.put(**new** Integer(125),**new** String("dinesh"));

m.put(**new** Integer(126),**new** String("anil"));

// m.put(new Integer(126),new String("Nilesh"));(duplicate key)

System.*out*.println("(Assembling)Map data is"+m);

System.*out*.println("De-Assembling the data");

Set s=m.entrySet();

Iterator itr=s.iterator();

**while**(itr.hasNext())

{

Map.Entry me=(Map.Entry)itr.next();

Object kobj=me.getKey();

Object vobj=me.getValue();

**if**(kobj **instanceof** Integer)

{

Integer io=(Integer)kobj;

**int** i=io.intValue();

System.*out*.println("Integer data(Key) is--->"+i);

}

**if**(vobj **instanceof** String)

{

String ss=vobj.toString();

System.*out*.println("String data(value) is--->"+ss+"\n");

}

}

}

}

**HashMap:**

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

public class Hashmapdemo

{

public static void main(String[] args)

{

HashMap m=new HashMap ();

System.out.println("Assembling the data");

m.put("AP","Hyderabad");

m.put("MH","Mumbai");

m.put("WB","calcutta");

m.put("TN","chennai");

m.put("Orissa","Bhuvaneswar");

m.put("Mijoram","Ijwal");

System.out.println("(Assembling)Map data is"+m);

System.out.println("De-Assembling the data");

Set s=m.entrySet();

Iterator itr=s.iterator();

while(itr.hasNext())

{

Map.Entry me=(Map.Entry)itr.next();

Object kobj=me.getKey();

Object vobj=me.getValue();

if(kobj instanceof String)

{

String ss=kobj.toString();

System.out.println("Integer data(Key) is--->"+ss);

}

if(vobj instanceof String)

{

String ss=vobj.toString();

System.out.println("String data(value) is--->"+ss+"\n");

}

}

}

}

**TreeMap :**

**import** java.util.\*;

**public** **class** Treemapdemo

{

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

{

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

m.put("AAA",**new** Integer(111));

System.*out*.println("Map object data is:"+m);

TreeMap obj=**new** TreeMap();

obj.put("S",**new** Integer(1));

obj.put("T",**new** Integer(2));

obj.put("U",**new** Integer(3));

obj.put("V",**new** Integer(4));

System.*out*.println(" TreeMap object data is"+obj);

obj.putAll(m);

System.*out*.println(" TreeMap object data after adding an object"+obj);

System.*out*.println(" First obj is "+obj.firstKey());

System.*out*.println(" Last obj is "+obj.lastKey());

System.*out*.println(" postion of V is"+obj.ceilingEntry("V"));

System.*out*.println(" object contains 4 value :"+obj.containsValue(4));

//De assembling

Set s=obj.entrySet();

Iterator itr=s.iterator();

**while**(itr.hasNext())

{

Map.Entry me=(Map.Entry)itr.next();

Object kobj=me.getKey();

Object vobj=me.getValue();

System.*out*.println("Kobj-------->"+kobj);

System.*out*.println("vobj--------->"+vobj);

}

obj.clear();

System.*out*.println(" TreeMap object data after clear an object"+obj);

}

}

**Vectordemo:-**

**import** java.util.\*;

**public** **class** vectordemo

{

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

{

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

v.add(**new** Integer(123));

v.addElement(**new** Integer(124));

System.*out*.println("vector object data is"+v);

Collection c=**new** ArrayList();

c.add(**new** Integer(125));

c.add(**new** Integer(126));

System.*out*.println("collection object data is"+c);

v.addAll(c);

System.*out*.println("vector object data after adding collection object is"+v);

//De assembling

Enumeration en=v.elements();

**while**(en.hasMoreElements())

{

Object obj=en.nextElement();

Integer io=(Integer)obj;

System.*out*.println("De assembling data is"+io.intValue());

}

}}

**Program on stack:**

import java.util.\*;

public class stackdemo

{

public static void main(String[] args)

{

Stack s=new Stack();

s.push(new Integer(1));

s.push(new Integer(2));

s.push(new Integer(3));

s.push(new Integer(4));

s.push(new Integer(5));

System.out.println("Stack data"+s);

System.out.println("Current position of cursor is "+s.peek());

//Apply pop()

s.pop();

System.out.println("Stack data after deletion"+s);

System.out.println("Current position of cursor after POP is "+s.peek());

//perform de-assembling operation

Enumeration en=s.elements();

while(en.hasMoreElements())

{

Object obj=en.nextElement();

Integer i=(Integer)obj;

System.out.println("Deassembling data is "+i.intValue());

}

}

}

**HashTable demo:**

import java.util.Enumeration;

import java.util.Hashtable;

import java.util.Map;

public class Hashtabledemo

{

public static void main(String[] args)

{

Hashtable ht=new Hashtable();

ht.put("Nag","java");

ht.put("xxx", "Testing");

ht.put("yyy","MS.net");

System.out.println("Values are:"+"\n");

System.out.println(ht.values());

//De assembling data

Enumeration en=ht.elements();

while(en.hasMoreElements())

{

System.out.println(en.nextElement());

}

}

}

**PropertiesDemo:**

**1.Preparation of properties file(student.prop)**

**sno=1**

**sname=harish**

**sloc=hyd**

**Source code:**

import java.util.\*;

import java.io.\*;

public class Propertiesdemo

{

public static void main(String[] args) throws FileNotFoundException,IOException

{

Properties p=new Properties();

FileInputStream fis=new FileInputStream("student.prop");

p.load(fis);

p.setProperty("fname","Anil");

System.out.println("fname is"+p.getProperty("fname"));

System.out.println("sno is"+p.getProperty("sno"));

}

}

**Dictionary :**

**import** java.util.\*;

**public** **class** Dictionarydemo

{

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

{

Dictionary d=**new** Hashtable();

d.put(**new** Integer(100),"Hundread");

d.put(**new** Integer(101),"Hundread and one");

d.put(**new** Integer(102),"Hundread and two");

System.*out*.println("Dictionary data is "+d);

Enumeration en=d.keys();

//De assembling the data

**while**(en.hasMoreElements())

{

System.*out*.println(en.nextElement());

System.*out*.println(en.toString());

}

}

}

**Date & calendar classes in java.util package:-**

**import** java.util.Calendar;

**public** **class** calendardemo

{

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

{

Calendar cl=Calendar.*getInstance*();

System.*out*.println(cl.getTimeInMillis());

System.*out*.println(cl.getFirstDayOfWeek());

System.*out*.println(cl.hashCode());

System.*out*.println(cl.isLenient());

System.*out*.println(cl.getTimeZone());

}

}

**Date class:**

**import** java.util.\*;

**public** **class** Datedemo

{

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

{

Date d=**new** Date();

System.*out*.println("Today date is"+d.toString());

System.*out*.println("Today date is"+d.~~getDate~~());

System.*out*.println("Day"+d.~~getDay~~());

System.*out*.println("Hour "+d.~~getHours~~());

System.*out*.println("Minutes"+d.~~getMinutes~~());

System.*out*.println("month"+d.~~getMonth~~());

System.*out*.println("year"+d.~~getYear~~());

System.*out*.println("seconds"+d.~~getSeconds~~());

}

}

**NavigableSet:**

The java.util.NavigableSet is nothing but an interface that is subtype of the java.util.SortedSet.

import java.util.\*;

import java.util.concurrent.\*;

public class NavigableSetExample{

public static void main(String[] args) {

System.out.println("Navigable set Example!\n");

NavigableSet <Integer>nSet = new ConcurrentSkipListSet<Integer>();

nSet.add(10);

nSet.add(20);

nSet.add(50);

nSet.add(30);

nSet.add(100);

nSet.add(80);

// Returns an iterator over the elements in navigable set,

in ascending order.

Iterator iterator = nSet.iterator();

System.out.print("Ascending order navigable set: ");

//Ascending order list

while (iterator.hasNext()){

System.out.print(iterator.next() + " ");

}

System.out.println();

//Descending order list

System.out.println("Descending order navigable set: " +

nSet.descendingSet() + "\n");

//Greater than or equal to the given element

System.out.println("Least element in Navigable set greater than

or equal to 35: " + nSet.ceiling(35));

//Less than or equal to the given element

System.out.println("Greatest element in Navigable set less than

or equal to 35: " + nSet.floor(35) + "\n");

//Viewing the portion of navigable set whose elements are

strictly less than the given element

System.out.println("Navigable set whose elements are strictly

less than '40': " + nSet.headSet(40));

//Viewing the portion of navigable set whose elements are

greater than or equal to the given element

System.out.println("Navigable set whose elements are greater

than or equal to '40': " + nSet.tailSet(40) + "\n");

//Removing first element from navigable set

System.out.println("Remove element: "+nSet.pollFirst());

//After removing the first element, now get navigable set

System.out.println("Now navigable set: " + nSet.descendingSet() + "\n");

//Removing last element from navigable set

System.out.println("Remove element: " + nSet.pollLast());

//After removing the last element, now get navigable set

System.out.println("Now navigable set: " + nSet.descendingSet());

}

}

**NavigableMap:-**

import java.util.\*;

import java.util.concurrent.\*;

public class NavigableMapExample{

public static void main(String[] args) {

System.out.println("Navigable Map Example!\n");

NavigableMap <Integer, String>navMap = new ConcurrentSkipListMap<Integer, String>();

navMap.put(1, "January");

navMap.put(2, "February");

navMap.put(3, "March");

navMap.put(4, "April");

navMap.put(5, "May");

navMap.put(6, "June");

navMap.put(7, "July");

navMap.put(8, "August");

navMap.put(9, "September");

navMap.put(10, "October");

navMap.put(11, "November");

navMap.put(12, "December");

//Displaying all data

System.out.println("Data in the navigable map: " + navMap.descendingMap()+"\n");

//Retrieving first data

System.out.print("First data: " + navMap.firstEntry()+"\n");

//Retrieving last data

System.out.print("Last data: " + navMap.lastEntry()+"\n\n");

//Retrieving the nreatest less than or equal to the given key

System.out.print("Nearest less than or equal to the given key: " + navMap.floorEntry(5)+"\n");

//Retrieving the greatest key strictly less than the given key

System.out.println("Retrieving the greatest key strictly less than the given key: " + navMap.lowerEntry(3));

//Retrieving a key-value associated with the least key strictly greater than the given key

System.out.println("Retriving data from navigable map greter than the given key: " + navMap.higherEntry(5)+"\n");

//Removing first

System.out.println("Removing First: " + navMap.pollFirstEntry());

//Removing last

System.out.println("Removing Last: " + navMap.pollLastEntry()+"\n");

//Displaying all data

System.out.println("Now data: " + navMap.descendingMap());

}

}

**Interveiw Questions on CollectionFrameWork in Java**

1.  What are limitations of object Arrays?

The main limitations of Object arrays are

* These are fixed in size ie once we created an array object there is no chance of increasing or decreasing size based on our requirement. Hence  If we don’t know size in advance , arrays are not recommended to use
* Arrays can hold only homogeneous elements.
* There is no underlying data structure for arrays and hence no readymade method support for arrays. Hence for every requirement programmer has to code explicitly. To over come  these problems collections are recommended to use

**Q2. What are differences between arrays and collections?**

|  |  |
| --- | --- |
| **Arrays** | **Collections** |
| 1.  Arrays r fixed in size and hence once we created an array we are not allowed to increase or decrease the size based on our requirement. | 1. Collections are growable in nature and hence based on our requirement we can increase or decrease the size. |
| 2.  Memory point of view arrays are not recommended to use | 2. Memory point of view collections are recommended to use. |
| 3. Performance point of view arrays are recommended to use | 3. Performance point of view collections are not recommended to use. |
| 4.  Arrays can hold only homogeneous elements | 4. Collections can hold both homogeneous and heterogeneous elements. |
| 5. Arrays can hold both primitives as well as objects | 5. Collections can hold only objects. |
| 6. For any requirement, there is no ready method support compulsory programmer has to code explicitly. | 6. For every requirement ready made method support is available. Being a programmer we have to know how to use those methods and we are not responsible to implement those. |

**Q3. what are differences between arrays and ArrayList?**  
  
**Refer  the answer of  Q2**

**Q4. What are differences between arrays and Vector?**  
  
**Refer the answer of Q2**

**Q5. What is Collection API ?**

It defines set of classes and interfaces which can be used for representing a group of objects as single entity

**Q6.  What is Collection framework?**

 It defines set of classes and inter faces which can be used for representing a group of objects as single entity

**Q7.  What  is difference between Collections and Collection?**

**Collection** is an interface which can be used for representing a group of individual objects as single entity and it acts as root interface of collection frame work.

**Collections**  is an utility class to define several utility methods for Collection implemented class objects.

**Q8.  Explain about Collection interface?**

* This interface can be used to represent a group of objects as a single entity.
* It acts as root interface for entire collection framework.
* It defines the most commonly used methods which can be applicable for any collection implemented class object

**Q9. Explain about List interface?**

List interface is a child interface of Collection interface. This can be used to represent group of individual objects in as a single entity where

* Duplicates are allowed
* Insertion order is preserved

**Q10.  Explain about Set interface?**

Set is a child interface of Collection interface. it can be used to represent a group of individual objects as a single entity where

* Duplicate objects are not allowed.
* Insertion order is not preserved

**Q11.  Explain about SortedSet interface?**

it**is child**interface of Set interface. it can be used to represent a group of individual objects in to a single entity where

* All the objects are arranged in some sorting order (Can be natural sorting order or customized).
* Duplicates are not allowed.

**Q12.  Explain about NavigableSet ?**

It is child interface of SortedSet and provides several utility methods for navigation purposes

* It doesn’t allows duplicates
* Insertion order is preserved
* It is introduced in 1.6 version

**Q13. Explain about Queue interface?**

If we want to represent a group of individual objects prior to processing, then we should go for Queue interface. It is child interface of Collection interface.It has introduced in 1.5 version.

**Q14. Explain about Map interface?**

Remember it is not a child Interface of Collection Interface and hence Map and Collection Interfaces doesn’t have any relationship.

* It can be used for representing a group of Objects as key, value pairs.
* Both keys and values should be objects
* Keys can t be duplicated but values can be duplicated.
* it has  introduced in 1.2 version

**Q15. Explain about SortedMap ?**

* If we want to represent a group of objects as key value pairs where all the entries are arranged according some sorting order of keys then we should go for SortedMap.
* It is child interface of Map.
* It has  introduced in 1.2 version

**Q16. Explain about NavigableMap?**

* It is child interface of SortedMap and defines several method for navigation purpose
* It is introduced in 1.6 version

**Q17.  Explain about ArrayList class?**

ArrayList is a Collection which can be used to represent a group of objects as a single entity.

* it is a implemented class for  List interface
* Introduced in 1.2 version
* The underlying data structure is resizable or growable array.
* Insertion order is preserved
* Duplicates are allowed
* Heterogeneous objects are allowed
* null insertion is possible
* This  class implements RandomAccess , Serializable , Cloneable interfaces
* Best choice  for retrieval purpose and worst if our frequent operation is insertion or deletion in the middle

**Q18. What is RandomAccess Interface?**

* If a collection class implements RandomAccess interface then we can access any of its element with the same speed.
* RandomAccess interface is marker interface and it dosent contains any methods.
* ArrayList and vector classes implements this interface.

**Q19. Explain about LinkedList class?**

LinkedList is a Collection implemented class which can be used for representing a group of objects as a single entity.

* LinkedList is the implemetation class for List interface
* Introduced in 1.2 version
* Underlying data Structure is   DoubleLinkedList
* Allows duplicates
* Insertion order is preserved
* Allows heterogeneous objects
* null insertion is possible
* LinkedList class implements Seriallizable and Cloneable interface but not RandomAccess interface
* Best choice  if frequent operation is insertion or deletion an objects in middle  but worst choice if frequent operation is retrieval.

**Q20. Explain about Vector class?**

Vector is a legacy collection class which can be used to represent a group of objects.

* Introduced in 1.0 version. it is legacy class
* The underlying data structure is resizable or growable array.
* Insertion order is preserved
* Duplicates are allowed
* Heterogeneous objects are allowed
* It is a implemented class for  List interface
* null insertion is possible
* Vector class implements RandomAccess ,Serializable,Cloneable interfaces
* Best Choice if frequent operation is retrieval and worst choice if frequent operation is insertion or deletion in the middle.
* All methods present in Vector class are synchronized hence Vector class object is thread safe.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1. No method is synchronized in the ArrayList class | 1. All methods in Vector are synchronized. |
| 2. ArrayList object is not thread safe. | 2.  Vector is thread safe. |
| 3. Relatively performance is high | 3. Relatively performance is low |
| 4. Introduced in 1.2 version and it is non legacy | 4. Introduced in 1.0 version and it is legacy |

**Q21. What is difference between ArrayList and Vector?**  
**Q22. How we can get synchronized version of ArrayList?**

Collections class contains synchronizedList() method for this Public static List synchronizedList(List l)  
**EX**  
                ArrayList l= new  ArrayList();  
                List l2=Collections.synchronizedList(l);

  Similarly we can get synchronized versions of Set and Map objects by the following methods.

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

**Q23. What is difference between size and capacity of a Collection Object?**

size means number  of objects present  where as capacity means no of objects it can accommodate.

**Q24. What are legacy classes and interfaces present  in Collections framework ?**

* Enumeration ---Interface
* Dictonary ------Abstract class
* Hashtable -----Concrete class
* Properties -----Concrete class
* Vector -----Concrete class
* Stack  -----Concrete class

**Q25. What is difference between ArrayList and Linked List?**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1. The underlying data structure is resizable or growable array. | 1. The underlying data structure is Double Linked List. |
| 2.  This is Best choice if frequent operation is retrieval and worst choice if frequent operation is insertion or deletion in the middle. | 2.  This is Best choice  if frequent operation is insertion or deletion in the middle and worst choice if frequent operation is retrieval . |
| 3. This class implements Serializable , Cloneable and RandomAccess interfaces. | 3. This class implements Serializable , Cloneable but not  RandomAccess interface. |

**Q26. what is difference Enumeration and Iterator?**

|  |  |
| --- | --- |
| **Enumeration** | **Iterator** |
| 1. It is legacy interface and introduced in 1.0 version | 1 It is non-legacy and introduced in 1.2 version |
| 2Applicable only for legacy classes and it is not universal cursor | 2Applicable for any Collection implemented class object. |
| 3While iterating the elements we are not allowed to remove the objects just we can perform only read operation | 3While iterating we can perform removal also in addition to read operation. |
| 4By using elements() method we can get Enumeration object | 4.   By using iterator() method we can get Iterator   object |

**Q27. What are limitations of Enumeration?**

* While iterating the elements we are not allowed to perform removal operation
* It is applicable only for legacy classes and it is not a universal cursor.
* It can retrieve the elements only in forward direction

**Q28. What is difference between enum and Enumeration?**

An **enum** can be used to define a group of named constants .It has  introduced in 1.5 version

**Ex:-**  
**Class**Beer{  
                KO,KF,RC,FO  
}

**Enumeration** is cursor to retrieve Objects one by one from Collection objects.

**Q29. What is difference between Iterator and ListIterator?**

* + ListIterator is the child interface of the Iterator
  + Iterator is the single direction cursor where as ListIterator is bidirectional cursor.
  + While iterating the elements by Iterator we can perform only read and remove operations. But by using ListIterator we can perform read,removal, replace and addition of new objects also.
  + Iterator is applicable for every Collecton implemented class object but ListIterator  is applicable only for List implemented class objects.
  + Iterator can be get by using iterator() of Collection interface where as ListIterator can be get by using listIterator() method of List interface
  + both are introduced in 1.2 version

**Q30. What is relation between ListIterator and Iterator?**  
      
      ListIterator is child interface of Iterator

**Q31. Explain about HashSet class?**

* The underlying data structure is Hashtable
* null values are accepted
* duplicates are not allowed
* insertion order is based on hashcode of the object hence insertion order is not preserved
* best  suitable if frequent operation is  search operations
* HashSet  class implements Serializable and Cloneable
* it is implementation class for Set interface
* heterogeneous objects are allowed
* it is introduced in 1.2 version

**Q32. If we are trying to insert duplicate values in Set what will happen?**

 If we are trying to insert duplicate objects to the HashSet  , we wont get any compile time or run time errors just the add(Object o) returns false and it doesn’t add that object.

**Q33. What is LinkedHashSet?**  
It is the child class of HashSet. The main difference between HashSet and LinkedHashSet is:In the case of HashSet insertion order is not preserved , but in the case of LinkedHashSet insertion will be preserved.

**Q34. Differences  between HashSet and LinkedHashSet?**

|  |  |
| --- | --- |
| **HashSet** | **LinkedHashSet** |
| 1The Underlying datastructure is Hashtable | 1The underlying datastructure is combination of LinkedList and Hashtable |
| 2Insertion Order is not preserved | 2     Insertion order is preserved. |
| 3Introduced in 1.2 version | 3     Introduced in 1.4 version |

**Q35. What are major enhancements in 1.4 version of collection frame work?**

LinkedHashSet  
 LinkedHashMap  
IdentityHashMap

**Q36. Explain about TreeSet?**  
  
 It is Collection object which can be used to represent a group of objects according to some sorting order.

* The underlying datastructure is Balanced tree
* Duplicates are not allowed
* All objects are stored according to some sorting order hence insertion order is not preserved
* Heterogeneous objects are not allowed violation leads to ClassCastException
* For an Empty TreeSet as firs element null value can be inserted but after inserting that first value if we are trying to insert any other objects then we will get NullPointerException
* For an non empty TreeSet if we are trying to  inser null value at run time u will get NullPointerException

**Q37. What are differences between List and Set interfaces?**

|  |  |
| --- | --- |
| **List** | **Set** |
| 1Insertion Order is preserved | 1Insertion Order is not preserved |
| 2Duplicate Objects are allowed | 2     Duplicate Objects are not allowed |
| 3The implemented classes are ArrayList,LinkedList , Vector and Stack classes | 3   The implemented classes are HashSet,            LinkedHashSet and Tree |

**Q38. What is Comparable interface?**

* This interface can be used for defining natural sorting order of the objects.
* It is present in java.lang package
* It contains a method public **int compareTo(Object obj1)**

**Q39. What is Comparator interface?**

* This interface can be used for implementing customized sorting order.It is present in java.util package
* It contains two methods
  + public int **compare**(Object ,Object)
  + public boolean **equals**(Object)

**Q40. What are differences b/t Comparable and comparator?**

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1This can be used for natural sorting order | 1This can be used for implementing customized sorting |
| 2This interface present in java.lang package | 2     This is present in java.util package |
| 3Contains only one method:  public int compareTo(Object obj1) | 3     It contains two methods.  public int compare(Object ,Object) public Boolean equals(Object) |
| 4    It is marker interface | 4  It is not a marker interface. |

**Q41. What is difference between HashSet and TreeSet?**

|  |  |
| --- | --- |
| **HashSet** | **TreeSet** |
| 1The underlying data structure is Hashtable | 1The underlying data structure is balanced tree |
| 2Heterogeneous objects are allowed | 2       Heterogeneous objects are not allowed  bydefalut |
| 3Insertion order is not preserved and it is based on hashcode of the objects | 3   Insertion order is not preserved and all the objects are inserted according to some sorting order. |
| 4null insertion is possible | 4   As the first element only null insertion is   possible and in all other cases we will get NullPointerException |

**Q42. What is Entry interface?**

It is inner interface of Map.   
In the Map each key value pair is considered as Entry object.  
   
**interface Map{**  
**//more code here**  
**interface Entry{**  
**Object  getKey()**  
**Object  getValue()**  
**Object  setValue(Object new)**  
**}**  
**}**

**Q43. Explain about HashMap?**

**A)**It is a Map Object which can be used used to represent a group of objects as key-value pairs.The underlying data structure is Hashtable

* Duplicate keys are not allowed duplicate values are allowed
* Insertion order is not preserved because insertion is based on hashcode of keys.
* Heterogeneous objects are allowed for  both keys and values
* null key is allowed  only once
* null values  are allowed multiple times
* Introduced in 1.2 version

**Q44. Explain about LinkedHashMap?**

It is child class of HashMap. It is exactly same as HashMap except the following difference.In the case of HashMap the insertion order is not preserved but in the case of LinkedHashMap insertion order is preserved. Introduced in 1.4 version

**Q45. Differences between HashMap and LinkedHashMap ?**

|  |  |
| --- | --- |
| **HashMap** | **LinkedHashMap** |
| 1.The underlying data structure is Hashtable | 1.The underlying data structure is a combination of Hashtable and linkedlist |
| 2.Insertion order is not preserved and it is based on hashcode of keys | 2    Insertion order is preserved |
| 3.Introduced in 1.2 version | 3   Introduced in 1.4 version. |

**Q46. Differences between HashMap and Hashtable?**

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1.The underlying data structure is Hashtable | 1.The underlying data structure of Hashtable |
| 2.No method is synchronized and hence HashMap object is not thread safe | 2 .All methods are synchronized and hence it is   thread safe |
| 3.Performance is high | 3.   Performance is low |
| 4.null insertion is possible for both keys and values | 4.   null insertion is not possible for both key and value violation leads to NullPointerException |
| 5.Introduced in 1.2 version and it is non legacy | 5.   Introduced in 1.0 version and it is legacy |

**Q47. What is IdentityHashMap?**

It is exactly same as HashMap except the following difference.In the HashMap JVM uses equals() method to identify duplicate keys  but in the  case of IdentityHashMap JVM uses == operator for this.

**Q48. What is difference between HashMap and IdentityHashMap?**  
        
Refer Q47 for the answer.

**Q49.WhatisWeakHashMap?**  
         
It is exactly same as HashMap except the following difference.In case of HashMap an Object is not eligible for garbage collection if it is associated with HashMap even though it dosent have any external references.  ie HashMap dominates garbage collector.But in case of WeakHashMap , if an Object is not having any external references then it is always eligible for garabage collectoion even though it is associated with weakHashMap.  ie  garbage collector dominates WeakHashMap

**Q50. What is difference between HashMap and WeakHashMap?**  
  Refer Q49 for the answer.

**Q51. What is TreeMap?**

 TreeMap can be used to store a group of objects as key-value pairs where all the entries are arranged according to some sorting order of keys.The underlying data structure is RED-BLACK Tree

* Duplicates keys are not allowed but values can be duplicated.
* Insertion order is not preserved because insertion is based on some sorting order
* If we are depending on Natural sorting order then keys should be homogeneous(violation leads to ClassCastException)  but values need not homogeneous
* In case of customized sorting order we can insert  heterogeneous keys and values
* For empty TreeMap as first entry with null values are allowed but after inserting that entry if we are trying to insert any other entry we will get NullPointerException
* For non empty TreeMap if we are trying to insert null keys we will get NullPointerException
* There are no restrictions for null values.

**Q52. What is Hashtable**  
        
Hashtable is a legacy Map and can be used to store objects as key value pairs.

* The underlying data sturucture is Hashtabe
* Duplicates keys are not allowed but duplicate values are allowed
* null insertion is not possible for both keys and values
* all methods are synchronized
* insertion order is not preserved because it is  based on hashcode  of keys
* heterogeneous Objects are allowed for both keys and values
* introduced in 1.0 version it is legacy class

**Q53. What is PriorityQueue?**

It represents a data structure to hold group of individual objects prior to processing based on some priority .it can be natural sorting order and it can be customized sorting order described by Comparator.  
It is the implementation class of Queue interface.

* Insertion order is not preserved because here insertion is done based on some sorting order
* Duplicates are not allowed
* null insertion is not possible even as first element also
* If we are depending on natural sorting order Objects should be homogeneous  violation leads to ClassCastException
* If we are depending on customized  sorting order Objects can be heterogeneous also.

**Q54. What is Arrays class?**

* It is utility class for arrays.
* It defines several utility methods for arrays like sorting an array or searching an element in array
* present in java.util package

**Q55. We are planning to do an indexed search in a list of objects. Which of the two Java collections should you use: ArrayList or LinkedList?**

1. ArrayList

**Q56. Why ArrayList is faster than Vector?**

All methods present in the Vector are synchronized  and hence  any method can be executed by only one thread at a time. It slows down the execution.But in ArrayList,  no method is synchronized and hence multiple thread are allowed execute simultaneously which speed up the execution.