## Arrays

* Arrays are used to represent group of elements as a single entity but these elements are homogeneous &fixed size.
* The size of Array is fixed it means once we created Array it is not possible to increase and decrease the size.
* Array in java is index based first element of the array stored at 0 index.  By using arrays possible to store primitive data & object data.

**Advantages of array:-**

* **Code optimization:**Instead of declaring individual variables we can declare group of elements by using array it reduces length of the code.
* **Flexibility:**We can store the group of objects easily & we are able to retrieve the data easily.
* **RandomAccess :** We can access the random elements present in the any location based on index.

First element

element at 5

th

position

0

1

2

3

4

5

6

7

8

index

L

ength is 9

There are two

approaches

to declare the array

10

20

30

40

50

60

70

80

90

**Approach 1:-** int[] a; **declaring** a = new int[5]; **instantiation**

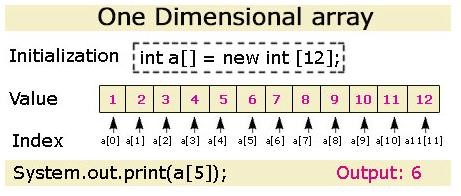
a[0]=10; **initialization**

a[1]=20;

**Approach 2:-** int a[]={10,20,30,40}; **//declaring, instantiation, intialization**

**Possible syntaxes:**

int[] values; int []values; int values[];



**Example :- different ways to print array data.** package com.dss;

public class Test

{ public static void main(String[] args)

{ int[] a = {10,20,30};

**//1-way**

System.out.println(a[0]);

System.out.println(a[1]);

System.out.println(a[2]);

**//2-way**

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

{ System.out.println(a[i]);

}

**//3-way**  for(int aa:a)

{ System.out.println(aa);

}

}

}

**Example** :

 when we create the array the array is created with default values later initialization is performaed. package com.dss; public class Test

{ public static void main(String[] args)

{ int[] a = new int[3]; for(int aa:a)

{ System.out.println(aa);

}

String[] s = new String[3]; s[0]="balu";

s[1]="anu";

for(String ss:s) { System.out.println(ss);

}

}

}

**Example** : finding null index vales.

String[] s = new String[3]; s[0]="balu";

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

{ if(s[i]==null)

System.out.println(i);

}

**Example:- adding the objects into Array and printing the object data.**

class Test

{ public static void main(String[] args)

{ Emp e1 = new Emp(111,"balu");

Emp e2 = new Emp(222,"anu");

Emp e3 = new Emp(333,"sravya"); Emp[] e = new Emp[5];

e[0]=e1; e[1]=e2; e[2]=e3;

for (Emp ee:e)

{ //System.out.println(ee); **in this line toString() executed prints hashcode**

System.out.println(ee.eid+”---”+ee.ename)

}

}

}

**Example:- printing array elements with elements and default values.**

class Test

{ public static void main(String[] args)

{ Emp[] e = new Emp[5]; e[0]=new Emp(111,"balu"); e[1]=new Emp(222,"anu"); e[2]=new Emp(333,"sravya"); for (Object e1:e)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | { | if (e1 instanceof Emp) |
|  |  |  | { Emp e2 = (Emp)e1; |
|  |  |  | System.out.println(e2.eid+"----"+e2.ename); |
|  |  |  | } |
|  |  |  | if (e1==null) |
|  |  |  | { System.out.println(e1); |
|  |  |  | } |
|  |  | } |  |
| } | } |  |  |

**Output:-** E:\>java Test

111----balu

222----anu 333----sravya

null null

 In above example if the location contains null value but we are trying to call the **e2.eid**JVM will generate NullPointerException so to overcome this problems use if-else condition to check the data.

**Example:-process of adding different types Objects in Object array**

**Emp.java: Student.java:-** class Emp class Student

{ int eid; String ename; { int sid; String sname;

Emp(int eid,String ename) Student(int sid,String sname)

{**//conversion of local to instance**  {**//conversion of local to instance**

this.eid=eid; this.sid=sid; this.ename=ename; this.sname=sname;

} }

} }

**Test.java**

class Test

{ public static void main(String[] args)

{ Object[] a= new Object[6]; a[0]=new Emp(111,"balu"); a[1]=new Integer(10); a[2]=new Student(1,"anu");

a[3]=new String(“balu”);

for (Object a1:a) { if (a1 instanceof Emp)

{ Emp e1 = (Emp)a1;

System.out.println(e1.eid+"---"+e1.ename);

}

if (a1 instanceof Student)

{ Student s1 = (Student)a1;

System.out.println(s1.sid+"---"+s1.sname);

}

if (a1 instanceof Integer)

{ System.out.println(a1);

}

if (a1 instanceof String)

{ System.out.println(a1);

}

if (a1==null)

{ System.out.println(a1);

}

}

}

}

**Example : Finding minimum & maximum element of the array** class Test

{ public static void main(String[] args) { int[] a = new int[]{10,20,5,70,4}; for (int a1:a) { System.out.println(a1);

}

**//minimum element of the Array**  int min=a[0];

for (int i=1;i<a.length;i++)

{ if (min>a[i])

{ min=a[i];

}

}

System.out.println("minimum value is ="+min);

**//maximum element of the Array**  int max=a[0];

for (int i=1;i<a.length;i++)

{ if (max<a[i])

{ max=a[i];

}

}

System.out.println("maximum value is ="+max);

}

}

**Example :- copy the data from one array to another array** class Test

{ public static void main(String[] args)

{ int[] copyfrom={10,20,30,40,50,60,70,80}; int[] copyto = new int[7]; System.arraycopy(copyfrom,1,copyto,0,7);

for (int cc:copyto) { System.out.println(cc);

}

}

}

**Example :- copy the data from one array to another array** class Test

{ public static void main(String[] args)

{ int[] copyfrom={10,20,30,40,50,60,70,80}; int[] newarray=java.util.Arrays.copyOfRange(copyfrom,1,4);

for (int aa:newarray)

{ System.out.println(aa);**//20 30 40**

}

}

}

**Example : To get the class name of the array** class Test

{ public static void main(String[] args)

{ int[] a={10,20,30};

System.out.println(a.getClass().getName());

}

}

**Example :- Taking array elements from keyboard by using scanner class.**

import java.util.\*;

class Test

{ public static void main(String[] args)

{ int[] a=new int[5];

Scanner s=new Scanner(System.in); System.out.println("enter values"); for (int i=0;i<a.length;i++)

{ System.out.println("enter "+i+" value"); a[i]=s.nextInt();

}

**//printing the data**

for (int a1:a) { System.out.println(a1);

}

}

}

**Example : Method parameter is array & method return type is array** class Test

{ static void m1(int[] a) **//method parameter is array**

{ for (int a1:a)

{ System.out.println(a1);

}

}

static int[] m2() **//method return type is array**

{ System.out.println("m1 method");

return new int[]{100,200,300};

}

public static void main(String[] args) { Test.m1(new int[]{10,20,30,40}); int[] x = Test.m2(); for (int x1:x) { System.out.println(x1);

}

}

}

**Example :- find the sum of the array elements.** class Test

{ public static void main(String[] args)

{ int[] a={10,20,30,40};

int sum=0;

for (int a1:a)

{ sum=sum+a1;

}

System.out.println("Array Element sum is="+sum);

}

}

**declaration of multi dimensional array:-** int[][] a; int [][]a; int a[][]; int []a[];

**Example :-** class Test

{ public static void main(String[] args)

{ int[][] a={{10,20,30},{40,50,60}};

System.out.println(a[0][0]);//10

System.out.println(a[1][0]);//40

System.out.println(a[1][1]);//50

}

}

1

0

30

20

10

30

10

20

0 1 2 0 1 2

**Example:-** class Test

{ public static void main(String[] args)

{ String[][] str={{"A.","B.","C."},{"balu","balu","balu"}};

System.out.println(str[0][0]+str[1][0]);

System.out.println(str[0][1]+str[1][1]);

System.out.println(str[0][2]+str[1][2]);

}

}

## Collectionsframework (java.util)

### Importance of collections:-

* The main objective of collections framework is to represent group of object as a single entity.
* In java Collection framework provide very good architecture to store and manipulate the group of objects.
* Collection API contains group of classes and interfaces that makes it easier to handle group of objects.
* Collections are providing flexibility to store, retrieve, and manipulate data.

### Arrays vs. Collections:-

Both Arrays and Collections are used to represent groupof objects as a single entity but the differences are as shown below.

#### Limitations of Arrays Advantages of Collections

1. Arrays are used to store homogeneous 1) Collections are used to store data(similar data). bothhomogeneous data.
2. Arrays are capable to store primitive & 2) Collections are capable to store only

Object type data object data.

1. Arrays are fixed in size, it means once we 3) Collections are growable in nature, it created array it is not possible to increase & means based on our requirement it decrease the size based on our requirement. is possible to increase & decrease
2. With respect to memory arrays are not the size.

recommended to use. 4) With respect to memory collections

1. If you know size in advance arrays are are recommended to use.

recommended to use because it provide 5) In performance point of view collections will give low performance good performance.

1. Arrays does not contains underlying Data compare to arrays.Collection classes contains

6) structure hence it is not supporting underlying data structure hence it

predefined methods. supports predefined methods.

Mainly the collections framework divided into two parts 1) Collection

**Collection**<interface>

**List**<interface> **Set**<interface> **Queue**<interface>

2) Map

* + All collection framework classes & interfaces are present in **java.util** package.
  + The root interface of the collection classes is : Collection
  + The root interface of the map classes is : Map
  + The parent interface of the Colletion is : Iterable (java.lang)

**The key interfaces of collection framework:**

* 1. Java.util.Collection
  2. Java.util.List
  3. Java.util.Set
  4. Java.util.SortedSet
  5. Java.util.NavigablaSet
  6. Java.util.Queue
  7. Java.util.Enumeration
  8. Java.util.Iterator
  9. Java.util.ListIterator
  10. Java.lang.Comparable --->java.lang package
  11. Java.util.Comparator

* 1. Java.util.Map
  2. Java.util.SotedMap
  3. Java.util.NavigableMap
  4. Map.Entry

#### Characteristics of Collection frame work classes:-

The collections framework contains group of classes but every class is used to represent group of objects as a single entity but characteristics are different. **1) The collect ion framework classes introduced Versions 2) Heterogeneous data allowed or not allowed.**

All most all collection framework classes allowed heterogeneous data except two classes

**i.** TreeSet ii. TreeMap

1. **Null insertion is possible or not possible.**
2. **Duplicate objects are allowed or not allowed.**

Inserting same object more than one time is called duplication. add(e1) add(e1)

1. **Insertion order is preserved or not preserved.**

In which order we are inserting element same order output is printed then say insertion order is preserved otherwise not.

**Input --->e1 e2 e3 output --->e1 e2 e3 insertion order is preserved Input --->e1 e2 e3 output --->e2 e1 e3 insertion order is not-preserved 6) Collection classes’ methods are synchronized or non-synchronized.**

If the methods are synchronized only one thread is allow to access, these methods are thread safe but performance is reduced.

If the methods are non-synchronized multiple threads are able to access, these methods are not thread safe but performance is increased.

All collection frame work classes are non-synchronized except  **Vector , HashTable 7) Collection classes underlying data structures.**

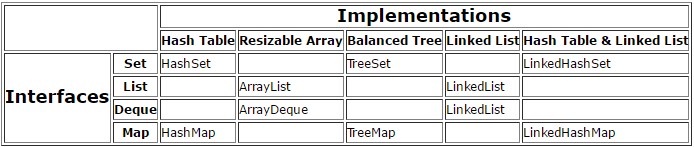
Every collection class contains underlying data structure hence it supports predefined methods.

**8) Collection classes supported cursors.**

The collection classes are used to represent group of objects as a single entity & To retrieve the objects from collation class we are using cursors**.** There are three direct sub interfaces of Collection interface.

**Collection vs. Collections:**

Collection is a root interface of collection frame work whereas Collections is utility class it contains methods to perform operations.



## List interface

**version**

**1.2**

**extends**

**version**

**1.2**

**implements**

**implements**

**implements**

**version**

**1.0**

**version**

**1.2**

**1.2**

**version**

**extends**

**1.0**

**version**

**Legacy classes(introduced in 1.0 version)**

**I**

**=**

**Interfacec**

**=**

**class**

Collection(i)

List(i)

ArrayList(c

LinkedList(c)

Vector(c)

Stack(c)

### List interface Implementation classes :-

1. ArrayList 3) Vector
2. LinkedList 4) Stack

### Legacy classes:-

The java classes which are introduced in 1.0 version are called legacy classes and **java.util** package contains 5 legacy classes.

1. HashTable4) Vector
2. Properties5) Dictionary <abstract class>
3. Stack6) Enumeration<interface>

By default all legacy classes methods are synchronized means thread safe.

**Java.util.ArrayList:-** To check parent classes and interface use below command.

**D:\balu>javap java.util.ArrayList**

public class java.util.ArrayList<E>

**extends** java.util.AbstractList<E>

**implements** java.util.List<E>,

java.util.RandomAccess, java.lang.Cloneable, java.io.Serializable

* Usually collection data is transferred from one JVM instance to other JVM instance. To support this requirement, every collection class must be inherited from java.io.Serializable interface.

Serializable is a marker interface providing serialization capabilities.

* Also, collection data can be copied. Hence, every collection class must be inherited from java.lang.Cloneable interface. Cloneable is a marker interface providing Cloneing capabilities.
* ArrayList and Vector classes implements java.util.RandomAccess, which is marker interface, to indicate that they support constant data access time.

### ArrayList:-

1. ArrayList Introduced in 1.2 version.
2. ArrayList stores Heterogeneous objects(different types).
3. In ArrayList it is possible to insert **Null** objects.
4. Duplicate objects are allowed.
5. ArrayList preserved Insertion order it means whatever the order we inserted the data in the same way output will be printed.
6. ArrayList methods are non-synchronized methods. 7) The under laying data structure is growable array.

**8)** By using cursor we are able to retrieve the data from ArrayList : **Iterator , ListIterator**

**Example:-**

**Case 1 : Up to 1.4 versions we must create wrapper class object then add that object into ArrayList.** import java.util.ArrayList; class Test

{ public static void main(String[] args)

{ rrayList al = new ArrayList();

al.add(new Integer(10));

al.add(new Character('c')); al.add(new Double(10.5));

System.out.println(al);

System.out.println(al.toString());

}

}

**Case 2: From 1.5 version onwards add the primitive data into ArrayList that data is automatically converted into wrapper object format is called Auto boxing.**

**Code before compilation:- Code after compilation:-** import java.util.ArrayList; iimport java.util.ArrayList;

class Test class Test

{ public static void main(String[] args) { public static void main(String args[]) { ArrayList al = new ArrayList(); { ArrayList arraylist = new ArrayList(); al.add(10); **//AutoBoxing**  arraylist.add(Integer.valueOf(10)); al.add('a'); arraylist.add(Character.valueOf('a'));

al.add(10.5);  arraylist.add(Double.valueOf(10.5));

System.out.println(al.toString()); System.out.println(arraylist);

}

} }

}

* In above example we are adding primitive value that primitive value is automatically converted into wrapper object format is called auto boxing.

* Whenever we are printing arraylist reference variable internally it calls toString() method on every object.

* toString() present in object class returns String representation of object(class-name@hashcode).

* String,StringBuffer,all wrapper classes toString() method overriding to return content of the object.

**Example-2**:-**working with object data.**

**Emp.java:- Student.java** class Emp class Student

|  |  |  |  |
| --- | --- | --- | --- |
| {          } | //instance variables  int eid;  String ename;  Emp(int eid,String ename)  { //conversion of local to instance this.eid=eid;  this.ename=ename;  } | {          } | //instance variables  int sid;  String sname;  Student(int sid,String sname)  { //conversion of local to instance this.sid=sid;  this.sname = sname;  } |

#### Test.java

import java.util.ArrayList; class Test

{ public static void main(String[] args)

{ Emp e1 = new Emp(111,"balu");

Student s1 = new Student(222,"xxx"); ArrayList al = new ArrayList();

al.add(10); al.add(e1);  al.add(s1);

al.add(“balu”);

System.out.println(al.toString());

for (Object o : al)

{ if (o instanceof Integer) {

System.out.println(o.toString());

}

if (o instanceof String) { System.out.println(o.toString());

}

if (o instanceof Emp){

Emp e = (Emp)o;

System.out.println(e.eid+"---"+e.ename);

}

if (o instanceof Student){

Student s = (Student)o;

System.out.println(s.sid+"---"+s.sname);

}

}

}

}

* Arrays are by default type safe it means the array contains only specific type of data.

Int array ---> stores only int data

String array ---> stores only String data

* Collections are not type safe (no guarantee on data) it means the collections contain different types of objects.

ArrayList al = new ArrayList(); al.add(10);

al.add(“balu”);

* If the collections are not type safe while reading the data we have to perform,
  1. Type casting
  2. Type checking

* To overcome above two problems to provide the type safety to the collections use generics.

* The generics are used to provide the type safety to the collections.

* When we provide type safety to the collection the advantages are , o No type checking.

o No type casting.

Normal version of arraylist used to store heterogeneous data.(but no type safety)

ArrayList al = new ArrayList(); al.add(10); al.add(“balu”);

Generic version of ArrayLisyt used to store only homogeneous data.(type safety)

ArrayList<String> al = new Arrraylist<String>();

al.add(“balu”);

al.add(“anu”);

**Collections are not type safe :- (problems) Generics provides type safety to the collections**

* 1. Type checking a. No type checking
  2. Type casting b. No type casting

**Example :-**

import java.util.ArrayList; public class Test {

public static void main(String[] args) {

int[] a={10,20,30};**//arrays are type safe**

for(int aa:a)

{ System.out.println(aa);

}

**//normal version of collection : no type safety**

ArrayList al = new ArrayList(); al.add(new Emp(111,"balu")); al.add(new Student(1,"xxx"));

for (Object o : al) { if (o instanceof Emp){

Emp e = (Emp)o;

System.out.println(e.eid+"---"+e.ename);

}

if (o instanceof Student){

Student s = (Student)o;

System.out.println(s.sid+"---"+s.sname);

}

}

**//generic version of collection : provides type safety**  ArrayList<Emp> a1 = new ArrayList<Emp>(); a1.add(new Emp(111,"balu")); a1.add(new Emp(222,"anu")); for (Emp e : a1)

{ System.out.println(e.eid+"---"+e.ename);

}

**//generic version of collection : provides type safety with null values**

ArrayList<Student> a2 = new ArrayList<Student>();

a2.add(new Student(1,"balu")); a2.add(new Student(2,"anu"));

a2.add(null);

for (Student s : a2)

{ if(s==null)

System.out.println(s);

else

System.out.println(s.sid+"---"+s.sname);

}

}

}

**Note :-** arrays are used to store the homogeneous data & collection generic version used to store homogeneous data but collections provides more flexibility with respect to the memory & operations.

**Example: - Basic operations on ArrayList**

import java.util.\*; class Test

{ public static void main(String[] args) { ArrayList al =new ArrayList();

al.add(10); al.add("balu"); al.add("anu"); al.add('a'); al.add(10);

al.add(null);

System.out.println("ArrayList data="+al);

System.out.println("ArrayList size-->"+al.size());

al.add(1,"durga"); //add the object at first index

System.out.println("ArrayList size-->"+al.size());

System.out.println("ArrayList Data="+al);

al.remove(1); //remove the object index base

al.remove("balu"); //remove the object on object base

System.out.println("arrayList size "+al.size());

System.out.println("ArrayList data="+al);

al.set(2,"xxx"); System.out.println(al);

System.out.println(al.isEmpty());

al.clear();

System.out.println(al.isEmpty());

}

}

**E:\>java Test**

ArrayList data=[10, balu, anu, a, 10, null] ArrayList size-->6

after adding objects ArrayList size-->7 ArrayList Data=[10, A1, balu, anu, a, 10, null] after removeing elemetns arrayList size 6 ArrayList data=[10, balu, anu, a, 10, null] false true

**observation:-**In above example when we remove the data by passing numeric value that is by default treated as index value.

ArrayList al = new ArrayList();

al.add(10);  **al.remove(10); // 10 is taken as index value : java.lang.IndexOutOfBoundsException:**

**Constructors to create ArrayList:- Constructor-1 public java.util.ArrayList();** new ArrayList();

The default capacity of the ArrayList is 10 once it reaches its maximum capacity then size is automatically increased by **New capacity = (old capacity\*3)/2+1 = 16**

**Constructor-2 public java.util.ArrayList(int user-capacity);** create ArrayList with initial capacity

ArrayList al = new ArrayList (20);

**New capacity = (old capacity\*3)/2+1 = 31**

**Constructor-3 public java.util.ArrayList(java.util.Collection<? extends E>);** Adding one collection data into another collection.

ArrayList a1 = new ArrayList(); a1.add(10);

ArrayList a2 = new ArrayList(a1); a2.add(20);

System.out.println(a2); //10 20

**Example :** There are two ways to add one collection data into another collection.

1. By using constructor approach.
2. By using addAll() method

* To add only one collection data into another collection use constructor approach.
* To add more than one collection into single collection use addAll() method.

import java.util.ArrayList; public class Test {

public static void main(String[] args) {

**//constructor approach**

ArrayList<String> a1 = new ArrayList<String>(); a1.add("balu");

ArrayList<String> a2 = new ArrayList<String>(a1);

a2.add("durga");

System.out.println(a2);

**//addAll() method to add the data**

ArrayList<String> b1 = new ArrayList<String>(); b1.add("aaa");

ArrayList<String> b2 = new ArrayList<String>(); b2.add("bbb");

ArrayList<String> b3 = new ArrayList<String>();

b3.addAll(b1); b3.addAll(b2); b3.add("ccc");

System.out.println(b3);

}

}

**Example:-** import java.util.\*;

class Test

{ public static void main(String[] args)

{ Emp e1 = new Emp(111,"balu");

Emp e2 = new Emp(222,"Sravya");

Emp e3 = new Emp(333,"aruna");

Emp e4 = new Emp(444,"anu");

ArrayList<Emp> a1 = new ArrayList<Emp>();

a1.add(e1); a1.add(e2);

ArrayList<Emp> a2 = new ArrayList<Emp>(); a2.addAll(a1); a2.add(e3); a2.add(e4);

System.out.println(a2.contains(e1)); System.out.println(a2.containsAll(a1));

a2.remove(e1);

System.out.println(a2.contains(e1));

System.out.println(a2.containsAll(a1));

**//printing the data**  for (Emp e:a2)

{ System.out.println(e.eid+"---"+e.ename);

}

}

}

**Example :-** a2.removeAll(a1); // it removes all **a1** data. a2.retainAll(a1); // it removes all **a2** data except **a1** **Example:Creation of sub ArrayList & swapping data :-**

Create sub ArrayList by using **subList(int,int)** method of ArrayList.

**public java.util.List<E> subList(int, int);**

To swap the data from one index position to another index position then use **swap()** method of Collections class. **public static void swap(java.util.List<?>, int, int);**

import java.util.\*; class Test

{ public static void main(String[] args)

{ ArrayList<String> a1 = new ArrayList<String>();

a1.add("balu"); a1.add("anu"); a1.add("Sravya"); a1.add("yadhu");

ArrayList<String> a2 = new ArrayList<String>(a1.subList(1,3));

System.out.println(a2); **//[anu,Sravya]**

System.out.println("before swapping="+a1);**//[balu, anu, Sravya, yadhu]**

Collections.swap(a1,1,3);

System.out.println("after swapping="+a1);// **[balu, yadhu, Sravya, anu]**

}

}

#### Example : ArrayList Capacity

import java.util.\*; import java.lang.reflect.Field; class Test

{ public static void main(String[] args)throws Exception { ArrayList<Integer> al = new ArrayList<Integer>(5);

for (int i=0;i<10 ;i++)

{ al.add(i);

System.out.println("size="+al.size()+" capacity="+getcapacity(al));

}

}

static int getcapacity(ArrayList l)throws Exception

{ Field f = ArrayList.class.getDeclaredField("elementData"); f.setAccessible(true); return ((Object[])f.get(l)).length;

}

}

|  |  |
| --- | --- |
| D:\>java Test size=1 capacity=5 size=2 capacity=5 size=3 capacity=5 size=4 capacity=5 size=5 capacity=5 | size=6 capacity=8 size=7 capacity=8 size=8 capacity=8 size=9 capacity=13 size=10 capacity=13 |

**Example :- conversion process**

import java.util.\*; class ArrayListDemo

{ public static void main(String[] args)

{

//**Conversion of array to ArrayList (by using asList() method)**

String[] str={"balu","Sravya","aruna"};

ArrayList<String> a1 = new ArrayList<String>(Arrays.asList(str));

a1.add("balu”); a1.add("anu");

for (String s: a1)

{ System.out.println(s);

}

//**Conversion of generic version of ArrayList to array by using toArray( T ) method**

ArrayList<String> a2= new ArrayList<String>();

a2.add("anu");

a2.add("Sravya");

String[] s = new String[al.size()]; s2.toArray(a);

for (String ss:s) { System.out.println(ss);

}

//**conversion of normal version ArrayList to Array by using toArray() method**

ArrayList a3= new ArrayList();

a3.add(10); a3.add('c');

a3.add("balu");

Object[] o = a3.toArray(); for (Object oo :o) { System.out.println(oo);

}

}

}

**java.util.LinkedList:**

public class java.util.LinkedList **extends** java.util.AbstractSequentialList

**implements** java.util.List<E>, java.util.Deque<E>, java.lang.Cloneable,java.io.Serializable.

1. Introduced in 1.2 version.
2. Heterogeneous objects are allowed.
3. Null insertion is possible.
4. Insertion order is preserved.
5. Linked List methods are non-synchronized.
6. Duplicate objects are allowed.
7. The under laying data structure is double linkedlist.
8. cursors :- Iterator,ListIterator**.**

**constructors:**

**LinkedList();** it builds empty LinkedList.

**LinkedList(java/util/Collection)** Used to add one collection data into another collection.

**Example:- LinkedList basic operations.**

import java.util.\*; class Test

{ public static void main(String[] args)

{ LinkedList<String> l=new LinkedList<String>();

l.add("B");

l.add("C");

l.add("D");

l.add("E");

l.addLast("Z");

l.addFirst("A");

l.add(1,"A1");

System.out.println("original content:-"+l);

l.removeFirst();

l.removeLast();

System.out.println("after deletion first & last:-"+l);

l.remove("E");

l.remove(2); **//remove the object of specified index**

System.out.println("after deletion :-"+l);//A1 B D

l.set(2,”balu"); **//set method used to replacement**

System.out.println("after seting:-"+l);

System.out.println(l.isEmpty());

l.clear();

System.out.println(l.isEmpty());

}

};

**Example:-Adding one collection data into another Collection.**

import java.util.\*; class Test

{ public static void main(String[] args)

{ ArrayList<String> al = new ArrayList<String>();

al.add("balu");

al.add("balu");

LinkedList<String> linked = new LinkedList<String>(al);

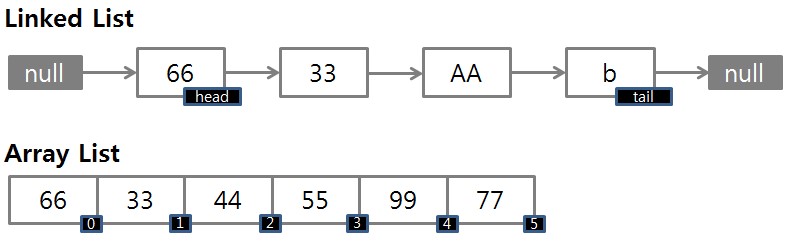
linked.add("anu"); linked.add("simran"); System.out.println(linked);

}

}

**ArrayList vs LinkedList:**

* ArrayList internally uses dynamic array to store the elements & LinkedList internally uses double linkedlist to store the elements.
* Manipulations on arraylist is slow because when we perform the operations internally it requires more shift operations. But manipulations on LinkedList is faster because shift operations are not required.
* With respect the memory Arraylist is recommended & with respect the operations LinkedList us recommended.
* ArrayList is implements RandomAccess interface hence the data read operations fast but LinkedList not implementing RandomAccess interface hence the data read operations are slow.
* RandomAccess is marker interface present in java.util package provides read operations capabilities.



**Vector:- (legacy class introduced in 1.0 version)** public class java.util.Vector **extends** java.util.AbstractList

**implements** java.util.List<E>,java.util.RandomAccess, java.lang.Cloneable,java.io.Serializable

1. Introduced in 1.0 version it is a legacy class.
2. Heterogeneous objects are allowed.
3. Duplicate objects are allowed.
4. Null insertion is possible.
5. Insertion order is preserved.
6. The under laying data structure is growable array.
7. Vector methods are synchronized.
8. Applicable cursors are Iterator,Enumeration,ListIterator.

Note : Vector is same as ArrayList but.

Vector methods are **synchronized.**

ArrayList methods are **non-synchronized.**

**constructors:-**

**Constructor 1:- public java.util.Vector();** new Vector();

The default initial capacity of the Vector is 10 once it reaches its maximum capacity it means when we trying to insert 11 element that capacity will become double[20 40 80….etc].

**Constructor 2:- public java.util.Vector(int user-capacity);**

Vector<String> vv = new Vector<String>(2); System.out.println(vv.capacity())**; //2**  vv.add("aaa"); vv.add("bbb");

System.out.println(vv.capacity()); **//4**

**Constructor 3:-public java.util.Vector(int user-capacity, int user-increment-value);**

Vector<String> v = new Vector<String>(2,5); System.out.println(v.capacity()); **//2**

v.add("balu");

v.add("aruna");

v.add("Sravya");

System.out.println(v.capacity()); **//7**

**Constructor 4:- public java.util.Vector(java.util.Collection):**Adding one collection data into another

ArrayList<String> al = new ArrayList<String>(); al.add("no1");

Vector<String> v = new Vector<String>(al); v.add("balu");

System.out.println(v);

**Example:-** import java.util.\*;

class Test

{ public static void main(String[] args)

{ Vector<Integer> v=new Vector<Integer>(); for (int i=0;i<5 ;i++ )

{ v.add(i);

}

Enumeration<Integer> e = v.elements(); while (e.hasMoreElements()) { Integer i = e.nextElement();

System.out.println(i);

}

System.out.println(v);

}

}

**Example :-** add the data in vector print even elements remove odd elements.

import java.util.Iterator; import java.util.Vector;

public class Test {

public static void main(String[] args) { Vector<Integer> v = new Vector<Integer>(2,7);

v.add(10);

v.add(20);

v.add(30);

v.add(5);

v.add(3);

System.out.println(v.capacity()); Iterator<Integer> itr = v.iterator(); while(itr.hasNext()) { Integer x = itr.next();

if(x%2==0)

{ System.out.println(x);

}

else

{ itr.remove();

}

}

System.out.println(v);

}

}

**Copying data from Vector to ArrayList:-** use **copy()** method of Collections class. import java.util.\*;

class Test

{ public static void main(String[] args)

{ ArrayList<String> al = new ArrayList<String>();

al.add("10"); al.add("20");

Vector<String> v = new Vector<String>();

v.add("ten");

v.add("twenty");

Collections.copy(al,v);

System.out.println(al);

}

}

**Stack:- (legacy class introduced in 1.0 version)**

1. It is a child class of vector.
2. Introduce in 1.0 versionit is a legacy class. 3) It is designed for LIFO(last in fist order ).

Example:- import java.util.\*;

class Test

{ public static void main(String[] args)

{ Stack<String> s = new Stack<String>();

s.push("balu"); **//insert the data top of the stack**

s.push("anu"); **//insert the data top of the stack**

s.push("durga");

System.out.println(s);

System.out.println(s.search("durga"));  **//1 last added object will become first**

System.out.println(s.size());

System.out.println(s.peek()); **//to return last element of the Stack**

s.pop(); **//remove the data top of the stack**

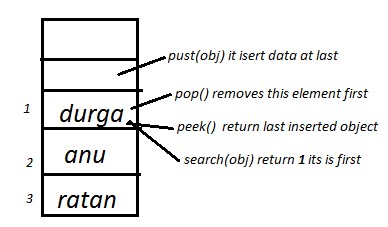
System.out.println(s); System.out.println(s.isEmpty());

s.clear();

System.out.println(s.isEmpty());

}

}



There are three ways to read the data from collection classes

1. By using for-each loop
2. By using get() method
3. By using cursor

There are three types of cursors in java

1. Enumeration
2. Iterator
3. listIterator

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Enumeration** | **Iterator** | **ListIterator** |
| **Purpose** | Read the data | Read the data | Read the data |
| **Is legacy?** | Yes 1.0 | No 1.2 version | No 1.2 version |
| **It is applicable for** | Only legacy classes | For all classes | Only for list classes |
| **Universal or not** | No | Yes | no |
| **Hot to get the object** | Using elements() method | Using iterator() method | Using listIterator() method |
| **Navigation** | Only forward | Only forward | Bi-directional |
| **Methods** | hasMoreElements(); nextElement(); | hasNext() next() , remove() | 9-methods |
| **Operations** | Only read | Read & remove | Read & remove  &update & add |
| **Class or interface** | Interface | Interface | Interface |
| **Normal & generic type** | Supports both | Support both | Support both |

**ListIterator methods:-** public abstract boolean hasNext(); public abstract E next(); public abstract boolean hasPrevious(); public abstract E previous(); public abstract int nextIndex(); public abstract int previousIndex(); public abstract void remove(); public abstract void set(E); **//replacement**  public abstract void add(E);

**Example :-** import java.util.\*;

class Test

{ public static void main(String[] args)

{ ArrayList<String> al =new ArrayList<String>();

al.add("balu"); al.add("anu");

al.add("sravya");

/**/1st appraoch to print Collection data : using for-each loop**

for (String a : al )

{ System.out.println(a);

}

**//2nd approach to print Collection data : using get() method**

int size = al.size(); for (int i=0;i<size;i++) { System.out.println(al.get(i));

}

**// 3rd approach : normal version of cursor**

Iterator itr1 = al.iterator(); while (itr1.hasNext())

{ String str =(String)itr1.next();

System.out.println(str);

}

**// 3rd approach : Generic version of cursor**  Iterator<String> itr2 = al.iterator(); while (itr2.hasNext()) { String str =itr2.next();

System.out.println(str);

}

}

}

**get() method to read the data :-**

**//normal version of arrayslit vs get() : type casting required**

ArrayList a1 = new ArrayList(); a1.add("balu");

String s1 = (String) a1.get(0);

System.out.println(s1);

**//normal version of arrayslit vs get() : type casting not required**

ArrayList<String> a2 = new ArrayList<String>(); a2.add("balu");

String s2 = a2.get(0);

System.out.println(s2);

**Example:-** import java.util.\*; class Test

{ public static void main(String[] args)

{ ArrayList<String> al =new ArrayList<String>();

al.add("balu"); al.add("anu");

al.add("sravya");

ListIterator<String> lstr = al.listIterator();

lstr.add("suneel"); while(lstr.hasNext())

{ String s = lstr.next();

if (s.equals("anu"))

{ lstr.set("Anushka");

}

if (s.equals("balu"))

{ lstr.remove();

}

}

System.out.println(lstr);

}

}

 Sunnel object is added at first position because when we create the iterator cursor the cursor is pointing to before first record.

**Example:-printing data in forward and backward directions.**

import java.util.\*; class Test

{ public static void main(String[] args)

{ ArrayList<String> al =new ArrayList<String>();

al.add("balu"); al.add("anu");

al.add("sravya");

ListIterator<String> lstr = al.listIterator();

System.out.println("printing data forward direction");

while(lstr.hasNext()) { String s = lstr.next()

System.out.println(s);

}

System.out.println("printing data backward direction");

while(lstr.hasPrevious()) { String s = lstr.next()

System.out.println(s);

}

}

}

**Cloning & serialization process :-**

In Collection frame work every class implements Cloneable & Serializable interfaces to support cloning process & serialization process.

**Example : Serialization process**

* To perform serialization of particular class that class must implements Serializable interface.
* To perform serialization of ArrayList inside the arraylist all objects must be serializable objects.

package com.dss; import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException; import java.io.ObjectInputStream; import java.io.ObjectOutputStream;

import java.util.ArrayList;

public class Test {

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

ArrayList<Emp> al = new ArrayList<Emp>(); al.add(new Emp(111, "balu")); al.add(new Emp(222, "anu"));

al.add(new Emp(333, "durga"));

**//Serialization process**

FileOutputStream outputStream = new FileOutputStream("abc.txt");

ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream); objectOutputStream.writeObject(al); outputStream.close();

objectOutputStream.close();

System.out.println("serialization process completed......");

**//Deserialization process**

FileInputStream inputStream = new FileInputStream("abc.txt");

ObjectInputStream objectInputStream = new ObjectInputStream(inputStream); ArrayList<Emp> arraylist = (ArrayList<Emp>)objectInputStream.readObject(); outputStream.close();

objectInputStream.close();

System.out.println("Deserialization process completed......");

for(Emp e:arraylist)

{ System.out.println(e.eid+"---"+e.ename);

}

}

}

**Example :** class Emp implements Serializable { }

class Student { } class Test

{ public static void main(String[] args) { ArrayList al = new ArrayList(); al.add(new Emp());

al.add(new Student());

}

}

* To perform serialization of ArrayList inside the arraylist all objects must be serializable objects.
* In above example serialization of Arraylist data is not possible because the Student obj is not implements seralizable interface.

**Example :- cloning process** import java.util.\*; class Test

{ public static void main(String[] args)

{ LinkedList<String> linked= new LinkedList<String>();

linked.add("First"); linked.add("Second"); linked.add("Third");

linked.add("Random");

System.out.println("Actual LinkedList:"+linked);

LinkedList<String> copy = (LinkedList) linked.clone();

System.out.println("Cloned LinkedList:"+copy);

}

}

E:\>java Test

Actual LinkedList:[First, Second, Third, Random]

Cloned LinkedList:[First, Second, Third, Random]

Ex:

import java.io.Serializable; import java.util.ArrayList; import java.util.RandomAccess;

public class Test {

public static void main(String[] args) {

ArrayList<String> al = new ArrayList<String>();

System.out.println(al instanceof Serializable);

System.out.println(al instanceof RandomAccess);

System.out.println(al instanceof Cloneable);

}

}

**Sorting data by using sort() method of Collections class:**

* It is possible to sort the collection data by using sort() method of Collections class and by default it perform ascending order .

* **if we want to perform sorting,** o **The data must be homogenous** o **Must implements Comparable interface.**

* In java String,all wrapper classes are implementing comparable interface by default.

* **To perform sorting internally JVM uses compareTo() method** o if you want perform sorting of two diff object JVM will generate ClassCastException.

o If you are perform sorting with null obj JVM will generate NullPointerExcpetion.

**Example :-**

import java.util.\*; class Test

{ public static void main(String[] args)

{ ArrayList<String> al = new ArrayList<String>();

al.add("balu"); al.add("anu");

al.add("Sravya");

System.out.println("ArrayList data before sorting="+al);

Collections.sort(al);

System.out.println("ArrayList data after sorting ascending order="+al);

}

}

**Case 1**: if we are trying to perform sorting of heterogeneous data , while performing comparison(by

using comapreTo() method) JVM will generate **java.lang.ClassCastException**

ArrayList al = new ArrayList(); al.add("balu"); al.add(10);

Collections.sort(al);

**Case 2:** when we perform sorting of data if the data contains null value while performing comparison (by using compareTo()) JVM will generate **java.lang.NullPointerException**.

ArrayList al = new ArrayList(); al.add("balu");

al.add(null);

Collections.sort(al);

* If we want to perform descending order use Collections.reverseOrder() method along with Collection.sort() method.

Collections.sort(list , Collections.reverseOrder());

**Java.lang.Comparable :**

* If we want to sort user defined class like Emp based on eid or ename with default natural sorting order then your class must implements Comparable interface.

* Comparable interface present in java.lang package it contains only one method compareTo(obj) then must override that method to write the sorting logics.

* If your class is implementing Comparable interface then that objects are sorted automatically by using **Collections.sort().** And the objects are sorted by using compareTo() method of that class.

“balu”.compareTo(“anu”) ==> +ve ==>change the order

“balu”.compareTo(“balu”) ==> 0 ==>no change

“anu”.compareTo(“balu”) ==> -ve ==>no change

**Ex :** Normal version of comparable performing sorting of eid **Emp.java:**

class Emp implements **Comparable**

{ int eid;

String ename;

Emp(int eid,String ename)

{ this.eid=eid;

this.ename=ename;

}

public int compareTo(Object o)

{ Emp e = (Emp)o; if (eid == e.eid )

return 0;

else if (eid > e.eid)

return 1; else return -1;

}

}

**Ex:** Generic version of Comparable performing sorting of ename

**Emp.java** class Emp implements **Comparable<Emp>**

{ int eid;

String ename;

Emp(int eid,String ename)

{ this.eid=eid; this.ename=ename;

}

public int compareTo(Emp e)

{ return ename.compareTo(e.ename);

}

}

**Test.java: use any one Emp.java file to perform sorting**

import java.util.\*;

class Test

{ public static void main(String[] args)

{ ArrayList<Emp> al = new ArrayList<Emp>(); al.add(new Emp(333,"balu")); al.add(new Emp(222,"anu")); al.add(new Emp(111,"Sravya")); Collections.sort(al);

Iterator itr = al.iterator(); while (itr.hasNext()) { Emp e = (Emp)itr.next();

System.out.println(e.eid+"---"+e.ename);

}

}

}

**Java.lang.Comparable vs java.util.Comparator:-**

**Property**

1. **Sorting logics**

1. **Sorting method**

1. **Method calling to perform sorting**

1. **package**

### 5. which type of sorting Comparable

1. Sorting logics must be in the class whose class objects are sorting.

1. **Int compareTo(Object o1)** This method compares this object with o1 object and returns a integer.Its value has following meaning **positive –** this object is greater than o1 **zero** – this object equals to o1 **negative –**this object

is less than o1

1. Collections.sort(List)

1. Java.lang

1. Default natual sorting order **Comparator**

1. Sorting logics in separate class hence we are able to sort the data by using dif attributes.

1. int compare(Object o1,Object o2)

This method compares o1 and o2 objects. and returns a integer.Its value has following meaning. **positive** – o1 is greater than o2  **zero –**o1 equals to o2  **negative**– o1 is less than o1

1. Collections.sort(List,

Comparator)

1. **Java.util**

1. **For customized sorting order.**

**Comparator sorting: Emp.java:-** class Emp

{ int eid;

String ename;

Emp(int eid,String ename)

{ this.eid=eid;

this.ename=ename;

}

}

**EidComp.java:- normal version of comparator : type casting required** import java.util.Comparator; public class EidComp implements Comparator{

public int compare(Object o1, Object o2) {

Emp e1 = (Emp)o1; Emp e2 = (Emp)o2; if(e1.eid==e2.eid) return 0; else if(e1.eid>e2.eid) return 1; else return -1;

}

}

**EnameComp.java:- generic version of comparator : type casting not required** import java.util.Comparator;

class EnameComp implements Comparator<Emp>

{ public int compare(Emp e1,Emp e2)

{ return (e1.ename).compareTo(e2.ename);

}

}

**Test.java:-** import java.util.\*;

class Test

{ public static void main(String[] args)

{ ArrayList<Emp> al = new ArrayList<Emp>(); al.add(new Emp(333,"balu")); al.add(new Emp(222,"anu")); al.add(new Emp(111,"Sravya")); Collections.sort(al,new EidComp()); Iterator<Emp> itr = al.iterator(); while (itr.hasNext())

{ Emp e = itr.next();

System.out.println(e.eid+"---"+e.ename);

}

}

}

**Sorting of emp name :**  Collections.sort(al,new EnameComp());

**Sorting of emp id :**  Collections.sort(al,new EidComp());

### Set interface

**v**

**1.2**

**extends**

**v**

**1.2**

**implements**

**implements**

**v**

**1.2**

**V**

**1.2**

**extends**

**extends**

**1.4**

**v**

**v**

**1.6**

**implements**

**1.2**

**v**

collection(i)

Set(i)

HashSet(c)

LinkedHashSet(c)

SortedSet(i)

NavigableSet(i)

TreeSet(c)

**List vs Set :- List allows duplicates & set duplicates not allowed.**

#### Java.util.HashSet:-

public class java.util.HashSet **extends** java.util.AbstractSet

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

**HashSet:-**

1. Introduced in 1.2 version.
2. Heterogeneous objects are allowed.
3. Duplicate objects are not allowed if we are trying to insert duplicate values then we won't get any compilation &Execution errors simply add method returns false .
4. Null insertion is possible but if we are inserting more than one null it return only one null value (because duplicates are not allowed).
5. The under laying data structure is HashTable.
6. Insertion order is not preserved it is based on the hash code of the object (hashing mechanism).
7. Methods are non-synchronized.
8. It supports only Iterator cursor to retrieve the data.

**Constructors:-**

1. **public HashSet();** it creates default HashSet.

new HashSet(); default capacity: 16 default fill ratio : 0.75

1. **Public HashSet(int user-capacity);**

New HashSet(10);

It create the HashSet by specified capacity. Default fill ratio 0.75

1. **Public HashSet(int capacity,float fillRatio);**  new HashSet(10,0.56);

It initialize both capacity and fillratio(also called as load factor).

1. **Public HashSet(java/util/Collection);** Adding one collection data into another collection data.

HashSet<String> h1 = new HashSet<String>(); h1.add("balu");

HashSet<String> h2 = new HashSet<String>(h1);

h2.add("no1");

System.out.println(h2);

**Example:**

import java.util.\*; class Test

{ public static void main(String[] args)

{ HashSet<String> h = new HashSet<String>();

h.add("A");

h.add("B");

h.add("C");

h.add("D");

h.add("D");

Iterator<String> itr = h.iterator(); while (itr.hasNext()) { String str = itr.next();

System.out.println(str);

}

}

}

**Example:-** import java.util.\*; class Test

{ public static void main(String[] args)

{ HashSet<String> h = new HashSet<String>();

System.out.println(h.add("D")); //true

System.out.println(h.add("D")); //false

System.out.println(h);

}

}

**Java.util.LinkedHashSet:-** public class java.util.LinkedHashSet **extends** java.util.HashSet

**implements** java.util.Set<E>,

java.lang.Cloneable,java.io.Serializable

1. Introduced in 1.4 version and It is a child class of HashSet.
2. Heterogeneous objects are allowed.
3. Duplicate objects are not allowed if we are trying to insert duplicate values then we won’t get any compilation &Execution errors simply add method return false.
4. Insertion order is preserved.
5. Null insertion is possible only once(because duplication is not possible).
6. The under laying data structure is LinkedList & hashTable.
7. Methods are non-synchronized.
8. It supports only Iterator cursor retrieve the data.

**Note : HashSet not preserved insertion order but LinkedHashset preserved insertion order.**

**Conastructors:-**

Public LinkedHashSet();

Public LinkedHashSet(java/util/Collection<? extends E>);

Public LinkedHashSet(int capacity);

Public LinkedHashSet(int capacity,float fillRatio);

**Example:-** import java.util.\*; class Test

{ public static void main(String[] args)

{ Set<String> h = new LinkedHashSet<String>();

h.add("A");

h.add("B");

h.add("C");

h.add("D");

h.add("D");

Iterator<String> itr = h.iterator(); while (itr.hasNext()) { String str = itr.next();

System.out.print(str);

}

}

}

**Example** : eliminating duplicates by using Set interface.

ArrayList<String> al = new ArrayList<String>(); al.add("anu"); al.add("anu");

LinkedHashSet<String> lh = new LinkedHashSet<String>(al); lh.add("aaa");

System.out.println(lh);

##### Java.util.TreeSet:-

public class java.util.TreeSet **extends** java.util.AbstractSet<E>

**implements** java.util.NavigableSet<E>,

java.lang.Cloneable, java.io.Serializable

1. TreeSet introduced in 1.2 version.
2. Heterogeneous data is not allowed.
3. Insertion order is not preserved but it sorts the elements in some sorting order.
4. Duplicate objects are not allowed.
5. Null insertion is possible only once.
6. TreeSet Methods are non-synchronized.
7. The underlying data Structure is Balanced Tree.
8. It supports Iterator cursor to retrieve the data.

**Constructors:-**

**TreeSet();** It will create empty TreeSet that will be sorted in ascending order.

**TreeSet(java/util/Collection** It creates the TreeSet with some collection data.

**TreeSet(java/util/Comparator);**  It will create empty TreeSet with comparator sorting orde.

**TreeSet(java/util/SortedSet<E>);** It builds the TreeSet that contains the elements of SortedSet.

**Case -1:** TreeSet<String> t=new TreeSet<String>();

t.add("balu");

t.add("anu");

System.out.println(t); **//[anu, balu]**

* When we insert the data in TreeSet, by default it prints the data in sorting order(ascending or alphabetical order) because it is implementing SortedSet interface.
* To perform the sorting internally it uses compareTo() method and it compare the two objects it returns int value as a return value.

**Case 2:-** TreeSet t=new TreeSet();

t.add("anu");

t.add(10); **// java.lang.ClassCastException**

System.out.println(t);

 TreeSet allows homogeneous data, if we are trying to insert heterogeneous data while performing sorting by using compareTo() JVM will generate **java.lang.ClassCastException (**becauseit is not possible to compare integer data with String) **.**

**Case 3:-** TreeSet t=new TreeSet();

t.add("balu");

t.add(null); **//java.lang.NullPointerException**

System.out.println(t);

* If the TreeSet contains data if we are trying to insert null value at the time of comparison JVM will generate  **//java.lang.NullPointerException.**
* In java any object with comparison of null it will generate  **java.lang.NullPointerException.**

**Example:-TreeSet customized Sorting Order. (constructor-3)** import java.util.\*;

class Fruit

{ public static void main(String[] args)

{ TreeSet<String> t = new TreeSet<String>(new MyComp());

t.add("orange");

t.add("banana");

t.add("apple"); System.out.println(t);

}

}

class MyComp implements Comparator<String>

{ public int compare(String s1,String s2)

{ return s1.compareTo(s2); //[apple, bananna, orange]

//return -s1.compareTo(s2); //[orange, bananna, apple]

}

};

**Example:- passing sortedset object to TreeSet constructor. (constructor-4)** import java.util.\*; class Sravya

{ public static void main(String[] args)

{ TreeSet<Integer> t=new TreeSet<Integer>();

t.add(20);

t.add(40);

t.add(10);

t.add(30);

System.out.println(t); **//10 20 30 40**

SortedSet s = t.headSet(30);

TreeSet tt = new TreeSet(s);

System.out.println(tt); **//10 20**

}

}

**Example:-Different possibilities of sorting order.** import java.util.\*; class Test

{ public static void main(String[] args)

{ TreeSet<Integer> t=new TreeSet<Integer>(new MyComp());

t.add(50);

t.add(20);

t.add(40);

t.add(10);

t.add(30);

System.out.println(t);

}

}

class MyComp implements Comparator { public int compare(Object o1,Object o2)

{ Integer i1 = (Integer)o1;

Integer i2 = (Integer)o2;

//check all possibilities by placing comments

//return i1.compareTo(i2);

//return -i1.compareTo(i2);

//return i2.compareTo(i1);

//return -i2.compareTo(i1);

}

}

Note : please comment the return statement check the output.

**Example :- program to insert StringBuffer data into TreeSet to perform sorting in alphabetical order.**

import java.util.\*; class Test

{ public static void main(String[] args)

{ TreeSet<StringBuffer> t = new TreeSet<StringBuffer>(new MyComp()); t.add(new StringBuffer("ccc"));

t.add(new StringBuffer("aaa"));

t.add(new StringBuffer("bbb"));

System.out.println(t);

}

}

class MyComp implements Comparator<StringBuffer> { public int compare(StringBuffer sb1,StringBuffer sb2)

{ String s1 = sb1.toString();

String s2 = sb2.toString(); //return s2.compareTo(s1);

return -s1.compareTo(s2);

}

};

**Example :- write a program to insert String & StringBuffer object into TreeSet perform sorting.**

import java.util.\*;

class Test

{ public static void main(String[] args)

{ TreeSet t = new TreeSet(new MyComp()); t.add("balu");

t.add(new StringBuffer("sravya"));

t.add("anu");

t.add(new StringBuffer("suneelbabu"));

t.add("sri");

System.out.println(t);

}

}

class MyComp implements Comparator { public int compare(Object o1,Object o2)

{ String s1 = o1.toString();

String s2 = o2.toString();

**//comment the lines check the output**

return s1.compareTo(s2); //return -s1.compareTo(s2);

//return s2.compareTo(s1);

//return -s2.compareTo(s1);

}

};

**Example: - Basic operations on TreeSet.**

|  |  |  |
| --- | --- | --- |
| public E lower(E); | it print lower object of specified object | |
| public E higher(E); | it print higher object of specified object | |
| public java/util/SortedSet<E> subSet(E, E); | it print subset | |
| public java/util/SortedSet<E> headSet(E); | it print specified object above objects | |
| public java/util/SortedSet<E> tailSet(E); | it print specified objects below values | |
| public E pollFirst(); | it print and remove first |  |
| public E pollLast();    import java.util.\*; class Test  { public static void main(String[] args) | it print and remove last. |  |
| { TreeSet<Integer> t=new TreeSet<Integer>(); | |
| t.add(50); t.add(20); t.add(40); t.add(10); | | t.add(30); |
| System.out.println(t); | | //10 20 30 40 50 |
| System.out.println(t.headSet(30)); | | //[10,20] |
| System.out.println(t.tailSet(30)); | | //[30,40,50] |
| System.out.println(t.subSet(20,50)); | | //[20,30,40] |
| System.out.println("last element="+t.last()); | | //50 |
| System.out.println("first element="+t.first()); | | //10 |
| System.out.println("lower element="+t.lower(50)); | | //40 |
| System.out.println("higher element="+t.higher(20)); | | //30 |
| System.out.println("print & remove first element="+t.pollFirst()); //10  System.out.println("print & remove last element="+t.pollLast()); //50  System.out.println("final elements="+t); //20 30 40  System.out.println(t.remove(30));  System.out.println("final elements="+t); //20 40 | | |

}

}

**E:\>java Test**

**[10, 20, 30, 40, 50]**

**[10, 20]**

**[30, 40, 50] [20, 30, 40] last element=50 first element=10 lower element=40 higher element=30 print & remove first element=10 print & remove last element=50 final elements=[20, 30, 40] TreeSet size=true final elements=[20, 40]**

**Queue interface:**

 Queue fallows FIFO(first in first out)

**Example :- in priorityQueue insertion order is not preserved.**

public abstract E remove(); // it removes the data

public abstract E poll();

It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. public abstract E element();//It is used to retrieves, but not remove, the head of this queue. public abstract E peek();

It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

import java.util.Iterator; import java.util.PriorityQueue;

public class Test {

public static void main(String[] args) {

PriorityQueue<String> pq = new PriorityQueue<String>();

pq.add("balu"); pq.add("anu"); pq.add("durga"); pq.add("sunny"); pq.add("xxx");

System.out.println(pq);

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

System.out.println(pq.poll());

System.out.println(pq);

pq.remove(); pq.remove("xxx");

System.out.println(pq);

Iterator<String> itr = pq.iterator(); while(itr.hasNext()) { String s = itr.next();

System.out.println(s);

}

}

}

Assignment : create the class Book with fields : id, name , author, quantity

Create the PriorityQueue add 3-book objects print the data

**Note** : if we are adding the data in priorityQueue the data must be homogeneous & must implements comparable interface.

### Map interface:-

**1.2**

**v**

**implements**

**implements**

**v**

**1.2**

**1.2**

**V**

**extends**

**extends**

**1.4**

**v**

**1.6**

**v**

**implements**

**v**

**1.2**

Map(i)

HashMap(c)

LinkedHashMap(c

SortedMap(i)

NavigableMap(i)

TreeMap(C)

* Map is used to store two objects at a time in the form of key value pairs. Here the key is object & value is object.
* The key value pair is known as entry, the map contains group of entries.
* In map the keys must be unique but values we can duplicate.

**Java.util.HashMap:-** public class java.util.HashMap **extends** java.util.AbstractMap

**implements** java.util.Map

java.lang.Cloneable,java.io.Serializable

* 1. introducedin 1.2 version.
  2. Heterogeneous data allowed.
  3. Underlying data Structure is HashTable.
  4. Duplicate keys are not allowed but values can be duplicated.
  5. Insertion order is not preserved it is based on hashcode.
  6. Null is allowed for key(only once)and allows for values any number of times.
  7. Every method is non-synchronized.

#### Constructors:-

**HashMap();** it creates default HashMap.

**HashMap(java/util/Mapvar);**  Adding one map data into another .

**HashMap(int capacity);** It creates the hashmap with specified capacity but the default capacity is **16.**

**HashMap(int capacity, float fillRatio);**

It creates the hashMap with specified capacity & fillRatio.(default capacity is 16 & default fill ratio 0.75) **Entry:-**

* The each and every key value pair is called **Entry.**The Map contains group of entries.
* Entry is sub interface of Map interface hence get the entry interface by using Map interface. interface **Map**

{ interface **Entry**

{ public abstract Object getKey(); public abstract Object getValue();

public abstract Object setValue();

}

}

* To get all the keys use keyset() method.

**public java/util/Set<K> keySet();**

* To get all the values use values() method.  **public java/util/Collection<V> values();**

* To get all the entries use entrySet() method.

**public java/util/Set<java/util/Map$Entry<K, V>> entrySet();**

**Example :**

package com.dss; import java.util.Collection; import java.util.HashMap; import java.util.Iterator; import java.util.Map.Entry; import java.util.Set;

public class Test {

public static void main(String[] args) {

HashMap<Integer, String> h = new HashMap<Integer,String>(); h.put(111, "balu");

h.put(222, "anu");

h.put(333, "durga"); System.out.println(h);

Set<Integer> s = h.keySet();

System.out.println(s);

Collection<String> s1 = h.values();

System.out.println(s1);

Set<Entry<Integer,String>> s2 = h.entrySet();

Iterator<Entry<Integer,String>> iterator = s2.iterator(); while(iterator.hasNext())

{ Entry<Integer, String> e = iterator.next();

System.out.println(e.getKey()+"----"+e.getValue());

}

}

}

**Java.util.LinkedHashMap:-** public class java.util.LinkedHashMap **extends** java.util.HashMap

**implements** java.util.Map

1. interdicted in 1.4 version
2. Heterogeneous data allowed.
3. Underlying data Structure is HashTable & linkedlist.
4. Duplicate keys are not allowed but values can be duplicated.
5. Insertion order is preserved.
6. Null is allowed for key(only once)and allows for values any number of times.
7. Every method is non-synchronized.

#### Constructors:-

**LinkedHashMap();** it creates default HashMap. Default capacity : 16 default fill ratio : 0.75  **LinkedHashMap(java/util/Map);** Used to add one map data into another map.

**LinkedHashMap(int user-capacity);** It creates the hashmap with specified capacity.

**LinkedHashMap(int user-capacity, float fillRatio);**  creates hashMap with specified capacity & fillRatio.

**Test.java:**

package com.dss; import java.util.HashMap; import java.util.Iterator; import java.util.Map.Entry; import java.util.Set;

public class Test {

public static void main(String[] args) {

HashMap<Emp, Student> h = new HashMap<Emp,Student>(); h.put(new Emp(111, "balu"),new Student(1, "aaa"));

h.put(new Emp(222, "anu"),new Student(2, "bbb"));

Set<Entry<Emp,Student>> s2 = h.entrySet();

Iterator<Entry<Emp,Student>> iterator = s2.iterator(); while(iterator.hasNext())

{ Entry<Emp, Student> e = iterator.next();

Emp ee = e.getKey();

System.out.println(ee.eid+"----"+ee.ename);

Student ss = e.getValue();

System.out.println(ss.sid+"--"+ss.sname);

}

for (Entry<Emp, Student> m:h.entrySet())

{ Emp e = m.getKey();

System.out.println(e.eid+"---"+e.ename);

Student s = m.getValue();

System.out.println(s.sid+"---"+s.sname);

}

}

}

There are two approaches to add one map data into another map

1. constructor approach 2. by using putAll() method

constructor approach is used to add only one Map data into another Map. putAll() method is used to add more than one map data into another Map.

import java.util.\*;

class Test

{ public static void main(String[] args)

{ **//constructor approach**

LinkedHashMap<Integer,String> h1 = new LinkedHashMap<Integer,String>();

h1.put(111,"balu");

LinkedhashMap<Integer,String> h2 = new LinkedHashMap<Integer,String>(h1);

h2.put(222,"anu");

for (Map.Entry m : h2.entrySet())

{ System.out.println(m.getKey()+"---"+m.getValue());

}

**//by using putAll() to add the data**

LinkedHashMap<Integer,String> h11 = new LinkedHashMap<Integer,String>();

h1.put(111,"balu");

LinkedHashMap<Integer,String> h22 = new LinkedHashMap<Integer,String>(); h2.put(222,"anu");

LinkedHashMap<Integer,String> h33 = new LinkedHashMap<Integer,String>();

h33.putAll(h11); h33.putAll(h22);

for(Entry<Integer, String> e :h3.entrySet())

{ System.out.println(e.getKey()+"--"+e.getValue());

}

}

}

#### Java.util.TreeMap:-

public class java.util.TreeMap **extends** java.util.AbstractMap

**implements** java.util.NavigableMap,java.lang.Cloneable, java.io.Serializable

1. This class is introduced in 1.2 version.
2. It allows homogeneous data if we are trying to insert heterogeneous data at runtime while perform sorting JVM will generate ClassCastException.
3. Duplicate keys are not allowed but values can be duplicated.
4. Insertion order is not preserved it is based on some sorting order of keys.
5. The underlying data structure is red-black trees.
6. For empty treeset it is possible to insert null key once, but if the treeset contains data if we are inserting null keys at runtime we will get NullPointerException but for the values any number of null values insertion possible.

**Constructors:-**

**TreeMap();** it will create empty treemap that will be sorted by using natural order of its keys.

**TreeMap(java/util/Comparator);** It create treemap that will be sorted by using customized sorting oder.

**TreeMap(java/util/Map);** adding one map data into another map

**TreeMap(java/util/SortedMa);** It creates the treemap by initializing sortedmap data.

**Observations of TreeMap:**

**Case1:-** TreeMap h = new TreeMap(); h.put(444,"balu");

h.put(222,"anu");

h.put(111,"aaa");

System.out.println(h);**//{111=aaa, 222=anu, 444=balu}**

In treemap when we insert the data that will be printed in sorting order based on key.

**Case 2:-** TreeMap h = new TreeMap(); h.put(444,"balu");

h.put("balu","aaa"); **//java.lang.ClassCastException**

System.out.println(h);

Treemap allows homogeneous data, if we are inserting heterogeneous data while performing sorting it will generate **java.lang.ClassCastException.**

**Case 3:-** TreeMap h = new TreeMap(); h.put(444,"balu");

h.put(null,"aaa"); **//java.lang.NullPointerException**

System.out.println(h);

If the treemap contains data then we are adding null value hence while performing sorting it will generate  **java.lang.NullPointerException(any object with comparisionof null it will generate NullPointerException )**

**Example:- TreeMap data sorting (Constructor-2)**

import java.util.\*;

class Test

{ public static void main(String[] args)

{ TreeMap h = new TreeMap(new MyComp()); h.put("balu",111);

h.put("anu",222);

h.put("zzzz",333);

System.out.println(h); //{zzzz=333, balu=111, anu=222}

}

}

class MyComp implements Comparator <String>

{ public int compare(String s1,String s2)

{ //place the comments check the output return s1.compareTo(s2);

//return s2.compareTo(s1);

//return -s1.compareTo(s2);

//return s2.compareTo(s1);

}

}

**Example : adding one map data into another map (constructor-3)**

TreeMap<Integer, String> t1 = new TreeMap<Integer,String>();

t.put(10,"balu");

t.put(20,"anu");

TreeMap<Integer, String> t2 = new TreeMap<Integer,String>(t1); t.put(10,"balu");

t.put(20,"anu");

System.out.println(t2);

**Example :- creating new TreeMap by passing sortedMap data (constructor-4)** import java.util.SortedMap; import java.util.TreeMap;

public class Test {

public static void main(String[] args) { TreeMap<Integer, String> t = new TreeMap<Integer,String>(); t.put(10,"balu");

t.put(20,"anu");

t.put(5,"durga");

t.put(3,"sunny"); System.out.println(t);

SortedMap<Integer, String> sm = t.subMap(5, 20);

TreeMap<Integer, String> tt = new TreeMap<Integer,String>(sm);

System.out.println(tt);

}

}

#### Example:-

import java.util.\*;

class Test

{ public static void main(String[] args)

{ TreeMap<Integer,String> h = new TreeMap<Integer,String>();

h.put(111,"balu");

h.put(222,"anu");

h.put(333,"aaa");

h.put(444,"aaa");

h.put(555,"ccc");

System.out.println(h);

h.remove(555);

System.out.println(h.firstEntry());

System.out.println(h.lastEntry());

System.out.println(h.firstKey());

System.out.println(h.lastKey());

System.out.println(h.lowerKey(222));

System.out.println(h.higherKey(222));

**//creation of new TreeMap by passing SortedMap data**

SortedMap s1 = h.headMap(333);

TreeMap t1 = new TreeMap(s1);

System.out.println(t1);

**//creation of new TreeMap by passing SortedMap data**

SortedMap s2 = h.tailMap(333);

TreeMap t2 = new TreeMap(s2);

System.out.println(t2);

}

}

|  |  |
| --- | --- |
| **Example :-** |  |
| **Ceiling()** | it return current provided value or greater value but if treemap does not contains same or greater value then it returns null . |
| **floor():-** | it returns current value or less value but if treemap does notcontains same value or less then it return null. |

**pollFirstEntry:-** it removes first entry & it prints that entry. **pollLastEntry():-** it removes last entry and it prints that entry.

import java.util.\*; class Test

{ public static void main(String[] args)

{ TreeMap h = new TreeMap(); h.put(111,"balu");

h.put(222,"anu");

h.put(444,"aaa");

System.out.println(h);

System.out.println(h.ceilingKey(222));

System.out.println(h.ceilingEntry(333));

System.out.println(h.floorKey(222));

System.out.println(h.floorEntry(333));

System.out.println(h.ceilingKey(666));

Map.Entry m1 = h.pollFirstEntry();

System.out.println(m1.getKey()+"---"+m1.getValue());

Map.Entry m2 = h.pollLastEntry();

System.out.println(m2.getKey()+"---"+m2.getValue());

System.out.println(h);

}

}

#### Java.util.HashTable:-

public class java.util.Hashtable **extends**  java.util.Dictionary

**implements**  java.util.Map,java.lang.Cloneable, java.io.Serializable

1. Introduced in the 1.0 version it’s a legacy class.
2. Heterogeneous data allowed for both key & value.
3. Duplicate keys are not allowed but values can be duplicated.
4. Every method is synchronized hence only one thread is allowed to access it is a Thread safe but performance is decreased.
5. Null is not allowed for both key & Value , if we are trying to insert null values we will get NullPointerException.
6. The underlaying datastructure is hashtable.

#### Constructors:-

**HashTable();**it creates default HashMap.

**HashTable (java/util/Map<? extends K, ? extends V> var);** it creates the HashMap by initializing the values specified in var.

**HashTable (int capacity);**

It creates the hashmap with specified capacity but the default capacity is **11.**

**HashTable (int capacity, float fillRatio);**

It creates the hashMap with specified capacity & fillRatio.

**Example:-** import java.util.Hashtable; import java.util.Collection; import java.util.Set; class Test

{ public static void main(String[] args)

{ Hashtable<Integer,String> h = new Hashtable<Integer,String>();

h.put(1,"one");

h.put(2,"two");

h.put(3,"three"); System.out.println(h);

System.out.println(h.get("1"));**//one**  System.out.println(h.isEmpty());

h.remove(3);

System.out.println(h.containsKey("1"));

System.out.println(h.containsKey("3"));

System.out.println(h.containsValue("one"));

System.out.println(h.size());

System.out.println(h.isEmpty());

h.clear();

System.out.println(h.isEmpty());

}

}

Ex:

import java.util.Hashtable; import java.util.Enumeration;

public class Test {

public static void main(String[] args) {

Enumeration names;

String key;

Hashtable<String, String> hashtable = new Hashtable<String, String>();

**// Adding Key and Value pairs to Hashtable**  hashtable.put("Key1","balu"); hashtable.put("Key2","anu");

hashtable.put("Key3","durga");

names = hashtable.keys(); while(names.hasMoreElements()) { key = (String) names.nextElement();

System.out.println("Key: " +key+ " & Value: " + hashtable.get(key));

}

}

}

#### Java.util.IdentityHashMap:-

public class java.util.IdentityHashMap **extends**  java.util.AbstractMap

**implements** java.util.Map,java.io.Serializable, java.lang.Cloneable

**It is same as hashmap except one difference,**

In case of Hashmap JVM will use equals( ) method to identify duplicate keys.(it performs content comparison)

In case of identityhashmap JVM will use **==**operator to identify the duplicate keys.(it perform reference comparison)

**Example:-** import java.util.\*; class Test

{ public static void main(String[] args)

{ **//equals() method to identify duplicate keys.**

HashMap<Integer,String> h = new HashMap<Integer,String>(); h.put(new Integer(10),"balu");

h.put(new Integer(10),"anu");

System.out.println(h);

**//== operator to identify duplicate keys.**

IdentityHashMap<Integer,String> h1 = new IdentityHashMap<Integer,String>(); h1.put(new Integer(10),"balu"); h1.put(new Integer(10),"anu");

System.out.println(h1);

}

}

**E:\>java Test**

**{10=anu}**

**{10=anu, 10=balu}**

**Java.util.WeakHashMap:-** public class java.util.WeakHashMap **extends** java.util.AbstractMap

**implements** java.util.Map

**WeakHashMap is same as HashMap except fallowing difference,**

If an object is associated with hashmap that object is not destroyed even though it does not contains any reference type.

But in case of weakhashmap if the object does not contains reference type that object iseligible for garbage collector even though it associated with weakhashmap.

|  |  |  |
| --- | --- | --- |
| **HashMap**  import java.util.\*; class A  { public String toString()  { return "A";  }  public void finalize()  {System.out.println("object destroyed");  }  };  class Test  { public static void main(String[] args)  { HashMap h = new HashMap(); A a= new A(); | }  }  **E:\>java Test**  **{A=balu}**  **{A=balu}** | h.put(a,"balu"); System.out.println(h); a=null; System.gc();  System.out.println(h); |
| **WeakHashMap**  import java.util.\*; class A | |
| { public String toString()  { return "A";  }  public void finalize()  {System.out.println("object destroyed");  }  };  class Test  { public static void main(String[] args)  {WeakHashMap h = new WeakHashMap();  A a= new A(); | }  }  **E:\>java Test**  **{A=balu}**  **{}** | h.put(a,"balu"); System.out.println(h);  a=null; System.gc();  System.out.println(h); |
| **object destroyed** | |

**Conversion of non-synchronized version to synchronized :**

 Generally collection class methods are non- synchronized by default but it is possible to get synchronized version of Collection classes.

To get synchronized version of List interface use fallowing Collections class static method **public static List synchronizedList(List l)**

ArrayList al = new ArrayList();

List l = Collections.synchronizedList(al);

To get synchronized version of Set interface use fallowing Collections class static method **public static Set synchronizedSet(Set s)**

HasSet h = new HashSet();

Set h1 = Collections.synchronizedSet(h);

To get synchronized version of Map interface use fallowing Collections class static method **public static Map synchronized Map(Map m)**

HashMap h = new HashMap();

Map m = Collections.synchronizedMap(h);

To get synchronized version of TreeSet use fallowing Collections class static method

**Collections.synchronizedSortedSet(SortedSet<T> s)**

TreeSet t = new TreeSet();

SortedSet s = Collections.synchronizedSortedSet(t);

To get synchronized version of TreeMap use fallowing Collections class static method

**Collections.synchronizedSortedMap(SortedMap<K,V> m)**

TreeMap t = new TreeMap();

SortedMap s = Collections.synchronizedSortedMap(t);

#### Java.util.Properties:-

* To get the flexibility of modifications use properties file.
* In standalone applications(JDBC) or web-applications(web sites) the data is frequently changing like,Database username,Database password,url,driver …etc
* in above scenario for every change must perform modifications in all .java files but it is complex.to overcome this problem use properties file.
* Properties file is a normal text file with .properties extension & it contains key=value formatted data but both key and value is string format.
* Once we done modifications on .properties file that modifications are reflected all the .java files.

#### abc.properties :- username = system

password = manager

**Test.java:-** import java.util.\*; import java.io.\*; class Test

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

{ **//locate properties file**

FileInputStream fis=new FileInputStream("abc.properties");

**//load the properties file by using load() method of Properties class** Properties p = new Properties();

p.load(fis);

**//get the data from properties class by using getProperty()**

String username = p.getProperty("username");

String password = p.getProperty("password");

**//use the properties file data**

System.out.println("DataBase username="+username);

System.out.println("DataBase password ="+password);

}

}

#### Collections

1. What is the main objective of collections?
2. What are the advantages of collections over arrays?
3. Collection frame work classes are present in which package?
4. By using collection framework classes is it possible to store primitive data?
5. What is the root interface of collection framework?
6. What is the root interface of Map?
7. List out implementation classes of List interface?
8. What is the parent interface of Collection interface?
9. What are the collection classes not allowed heterogeneous data?
10. Can you please tell me some of the legacy classes present in collection framework?
11. What do you mean by auto-boxing?
12. Arrays are type safe or not?
13. How to provide type safety to the collection?
14. What is the purpose of generic version of collection classes?
15. What is tsshe difference between general version of ArrayList and generic version of ArrayList?
16. What is purpose of generic version of ArrayList & arrays?
17. What is the difference between normal version of collections & generic version of collection?
18. What are the ways to add one Collections data into another collection?
19. What is the difference between removeAll() & retainsAll()?
20. When we get the IndexOutOfBoundsException?
21. When we get the ArraysIndexOutOfBoundsException?
22. When we will get StringIndexOutOfBoundsException?
23. How to convert Collection data to arrays & Arrays data to collection?
24. How to convert arrays to collections & collections to arrays?
25. What is the difference between ArrayList and LinkedList?
26. How to decide when to use ArrayList and when to use LinkedList?
27. What is the difference between ArrayList & vector?
28. Arrays used to hold homogeneous data but what is the purpose of generic version of Collection?
29. What is the purpose of RandomAccess interface and it is marker interface or not?
30. All collection classes are commonly implemented Serializable & Cloneable what is the purpose?
31. What do you mean by cursor and how many cursors present in java?
32. How many ways are there to retrieve objects from collections classes what are those?
33. What is the purpose of Enumeration cursor and how to get that cursor object?
34. What are the cursors used to retrieve the objects forward & backward direction?
35. What is the purpose of Iterator and how to get Iterator Object?
36. What is the purpose of ListIterator and how to get that object?
37. What is the difference between Enumeration vs Iterator Vs ListIterator?
38. What are the applicable cursors on ArrayList,Vector,LinkedList?
39. List out implementation classes of set interface?
40. What is the difference between HashSet & linkedHashSet?
41. What is the purpose of TreeSet class?
42. What is the difference between Set & List interface?
43. How to perform the sorting of collection data?
44. To perform the sorting what are the conditions?
45. To perform the default sorting it internally uses which method?
46. What is the difference between collection & collections?
47. What is the difference between comparable & comparator?
48. When we will get ClassCastException?
49. When we will get the NullPointerException?
50. In Comparable by using which method we are writing sorting logics?
51. Is it possible to compare two different objects?
52. What is the difference between compreTo() & equals() method?
53. In Comparator by using which method we are writing sorting logics?
54. What is the purpose of Map interface?
55. List out implementation classes of map interface?
56. What do you mean by entry.
57. How to get all values objects , key objets,entry objects?
58. What is the difference between HashMap & LinkedHashMap?
59. How many ways are there to add one map data into another Map?
60. What is the difference between TreeSet and TtreeMap?
61. What is the difference between IdentityHashMap & WeakHashMap?
62. What is the difference between HashTable and Properties file key=value pairs?
63. HashTable null values are allowed or not?
64. What do you mean by properties file and what are the advantages of properties file?
65. By using properties file we can declare different type of data or not?
66. What is the difference between HashTable & HasMap?
67. How to convert non-synchronized version of method into synchronized version method?