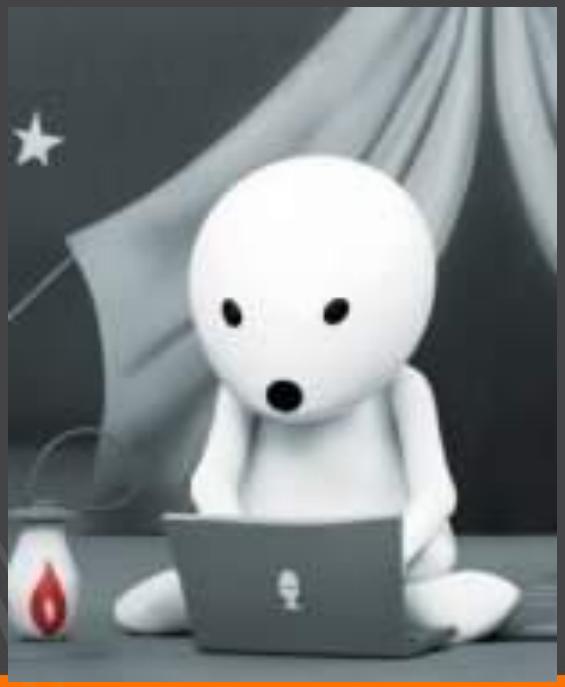
Object Oriented Programming

Unit 8



Collection Framework

Generics: **Collections,** Bounded types. Overview of collection framework: Need for collection framework in java, Vector and stack, Stack Iterators: Iterable and Iterator, Iterators, Using iterators in Java Utility Class: Collections utility class Arrays utility class: Arrays utility class, Java Arrays Sort Set: The Set interface and common implementations, Implement ArrayAsList, Implementations Queue: Queue, General-Purpose Queue Implementations Legacy Collections: Hashtable and Properties, Reading from properties file using Properties class, Comparator

Collection

- The collections framework is a **unified architecture for representing and manipulating collections**, enabling them to be manipulated independently of the details of their representation.
- It **reduces programming effort** while increasing performance. It enables **interoperability among unrelated APIs**, reduces effort in designing and learning new APIs, and fosters software reuse.
- The framework is based on more than a dozen collection interfaces. It includes implementations of these interfaces and algorithms to manipulate them.
- The **Collection interface** is at the **root of the collection hierarchy**. Sub interfaces of Collection include List, Queue, and Set.

Generics

- Generics allow type to be a parameter to methods, classes and interfaces.
  Syntax

  BaseType <Type> obj = new BaseType <Type>()

- In parameter type we cannot use primitives like 'int', 'char' or 'double'.

Iterator

An **Iterator is an object** that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the **java.util package**.

```
Iterator itr=list.iterator();

while(itr.hasNext())

{

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

}
```

boolean hasNext()

Returns true if the iteration has more elements. (In other words, returns true if next() would return an element rather than throwing an exception.)

next()

Returns the next element in the iteration.

ArrayList with Iterator

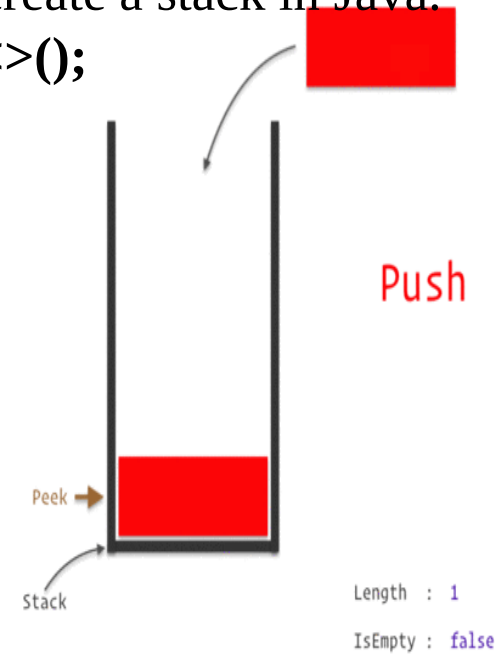
```
import java.util.*;

public class CollectionsProg {
   public static void main(String args[]) {
        ArrayList<String> list=new ArrayList<String>();       //Creating arraylist
        list.add("OOP");//Adding object in arraylist
        list.add("DBMS");
        list.add("CN");
        list.add("DS");
        //Traversing list through Iterator
        Iterator itr=list.iterator();
        while(itr.hasNext())
        {
        System.out.println(itr.next());
        }
    }
}
```

• Vector	• Stack Class	• ArrayList Class
• **Java Vector class** comes under the **java.util** package.	• E**xtends the Vector class.** • In stack, elements are stored and accessed in **Last In First Out** manner. • Import the **java.util.Stack** package	• Implements the **List interface**. • Before using ArrayList, we need to import the java.util.ArrayList package first.
• Vector **implements a dynamic array** that means it can grow or shrink as required.	• Grows and shrinks its size as needed when new elements are added or removed.	• Uses a **dynamic array to store the duplicate element of different data types**.
• Vector is **synchronized**	• Is **synchronized**	• Is **non-synchronized**
• **Vector v1=new Vector();** • **Vector v2=new Vector(4);** • **Vector v1=new Vector(3,6);**	**Stack<Type> stacks = new Stack<>();**	**ArrayList<Type> arr= new ArrayList<Type>();**

Stack

- The Java **collections framework has a class named Stack** that provides the functionality of the stack data structure.
- The Stack class **extends the Vector class.**
- In stack, elements are stored and accessed in **Last In First Out** manner. That is, elements are added to the top of the stack and removed from the top of the stack.
- In order to create a stack, we must import the **java.util.Stack** package first. Once we import the package, here is how we can create a stack in Java.

Stack<Type> stacks = new Stack<>();



```
import java.util.Stack;
public class StackDemo {
    public static void main(String args[]) {
        Stack<String> stackOfCards = new Stack<>();
        stackOfCards.push("Jack");
        stackOfCards.push("Queen");
        stackOfCards.push("King");
        stackOfCards.push("Ace");
        System.out.println("Stack => " + stackOfCards);
        System.out.println();

        String cardAtTop = stackOfCards.pop();
        System.out.println("Stack.pop() => " + cardAtTop);
        System.out.println("Current Stack => " + stackOfCards);
        System.out.println();

        cardAtTop = stackOfCards.peek();
        System.out.println("Stack.peek() => " + cardAtTop);
        System.out.println("Current Stack => " + stackOfCards);
    }}
```

```
Stack => [Jack, Queen, King, Ace]


Stack.pop() => Ace

Current Stack => [Jack, Queen, King]


Stack.peek() => King

Current Stack => [Jack, Queen, King]
```

Vectors

- **Java Vector class** comes under the **java.util** package.
- The vector class **implements a growable array of objects**. Like an array, it contains the component that can be **accessed using an integer index.**
- Vector is very useful if we don't know the size of an array in advance or we need one that can change the size over the lifetime of a program.
- Array is a set of similar data types which has **static memory allocation.**
- Vector **implements a dynamic array** that means it can grow or shrink as required. It is similar to the ArrayList, but with two differences-
  - Vector is **synchronized.**
  - The vector contains many legacy methods that are not the part of a collections framework
- The signature of the class is:

public class Vector<E>

extends Object<E>

implements List<E>, Cloneable, Serializable

Vectors(Contd.)

- **Constructors:**

Sr.No.	Constructor & Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10.
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4	**Vector(Collection c)** This constructor creates a vector that contains the elements of collection c.

Vectors(Contd.)

- **Example:** **Vector v1=new Vector();**

- It creates an empty Vector with the **default initial capacity of 10.** It means the Vector will be re-sized when the 11th element needs to be inserted into the Vector. Note: By default vector **doubles its size**. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

- **Example:** **Vector v2=new Vector(3);**

- It will create a Vector of initial capacity of 3.

- **Example:** **Vector v3=new Vector(4,6);**

- Here we have provided two arguments. The initial capacity is 4 and capacity increment is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

Vectors(Contd.)

- **void addElement(Object element):** It inserts the element at the end of the Vector.
- **void add(int index, Object element)**: Inserts the specified element at the specified position in this Vector.
- **int capacity():** This method returns the current capacity of the vector.
- **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
- **Object elementAt(int index):** It returns the element present at the specified location in Vector.
- **Object firstElement():** It is used for getting the first element of the vector.
- **Object lastElement():** Returns the last element of the vector.
- **boolean isEmpty():** This method returns true if Vector doesn't have any element.
- **boolean removeElement(Object element):** Removes the specifed element from vector.

Vectors(Contd.)

- **boolean removeAllElements():** It Removes all those elements from vector and size becomes zero.
- **int size():** It returns the current size of the vector.
- **void setSize(int size):** It changes the existing size with the specified size.
- **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
- **Object elementAt(int index):** It returns the element present at the specified location in Vector.
- **Object get(int index):** Returns the element at the specified index.
- **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.

```java
import java.util.*;
public class VectorAddExample
{
    public static void main(String args[])
    {
      Vector<String> vec = new Vector<String>(4);
      vec.add("Tiger");
      vec.add("Lion");
      vec.add("Dog");
      vec.add("Elephant");
      System.out.println("Vector element is: "+vec);
      vec.addElement("Rat");
      vec.addElement("Cat");
      vec.addElement("Deer");
      System.out.println("Vector elements are: "+vec);
       if(vec.contains("Tiger"))
        {
          System.out.println("Tiger is present at the index " +vec.indexOf("Tiger"));
        }
        else
        {
          System.out.println("Tiger is not present in the list.");
        }
     System.out.println("Third animal of the vector is = "+vec.elementAt(3));
     System.out.println("Get animal of the vector = "+  vec.get(4));
      System.out.println("Vector is empty = "+vec.isEmpty());
      System.out.println("The first animal of the vector is = "+vec.firstElement());
      System.out.println("The last animal of the vector is = "+vec.lastElement());
    }
}
```

```
C:\Users\Virendra\Desktop\Core Java>java VectorAddExample
Vector element is: [Tiger, Lion, Dog, Elephant]
Vector elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
Tiger is present at the index 0
Third animal of the vector is = Elephant
Get animal of the vector = Rat
Vector is empty = false
The first animal of the vector is = Tiger
The last animal of the vector is = Deer
```

ArrayList

- The ArrayList class implements the List interface.

- It uses a **dynamic array to store the duplicate element of different data types**. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

- The ArrayList class of the Java collections framework provides the functionality of **resizable-arrays.**

- Before using ArrayList, we need to import the java.util.ArrayList package first. Here is how we can create arraylists in Java:

ArrayList<Type> arrayList= new ArrayList<Type>();

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection<? extends E> c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

void add(int index, E element)
boolean add(E e)
boolean addAll(Collection<? extends E> c)
boolean addAll(int index, Collection<? extends E> c)
void clear()

int indexOf(Object o)
E remove(int index)
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean removeIf(Predicate<? super E> filter)
protected void removeRange(int fromIndex, int toIndex)
void replaceAll(UnaryOperator<E> operator)
void retainAll(Collection<?> c)

ArrayList Functions

```
ArrayList<String> al=new ArrayList<String>();
al.add("Mango");
al.add("Apple");
al.add("Banana");
al.add("Grapes");
 //accessing the element
 System.out.println("Returning element: "+al.get(1));
//changing the element
al.set(1,"Dates");
System.out.println(fruit);
//Traversing list
for(String fruit:al)
        System.out.println(fruit);
```

Output:

```
Returning element:
Apple
[Mango, Dates,
Banana, Grapes]
Mango
Dates
Banana
Grapes
```

Assigning values in ArrayasList

```
String x[]={"1","2","3","4","5","6","7","8","9"};

ArrayList<Integer> alit=new
ArrayList<Integer>(Arrays.asList(1,2,3,4,5,6,7,8,9,10,11));

ArrayList<String> alit2=new ArrayList<String>(Arrays.asList(x));

alit.size();     //To fetch the length
```

Set Interface	Queue Interface	Hashtable Class
-C**ollection that cannot contain duplicate elements**. Used to model the mathematical set abstraction.	- Present in **java.util package** extends the **Collection interface** - is used to hold the elements about to be processed in **FIFO (First In First Out) order.**	**- Class** implements a **hashtable**, which maps **keys to values**. It implements the **Map interface**. Contains unique elements.
Set<Integer> set = new HashSet<Integer>();	**Queue<String> pq = new PriorityQueue<>();**	**Hashtable<Type1,Type2> hm =new Hashtable<Type1,Type2>();**
Is non synchronized	**Is synchronized**	**Is synchronized**
Implementation of Set: **HashSet, TreeSet and LinkedHashSet**	--	--

Queue

```
Queue<Integer> q = new LinkedList<>();
for (int i=0; i<5; i++)
q.add(i);
System.out.println("Elements of queue-"+q);
int removedele = q.remove();
System.out.println("removed element-" + removedele);
System.out.println(q);
 int head = q.peek();
System.out.println("head of queue-" + head);
int size = q.size();
System.out.println("Size of queue-" + size);
```

```
Elements of queue-[0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
Size of queue-4
```

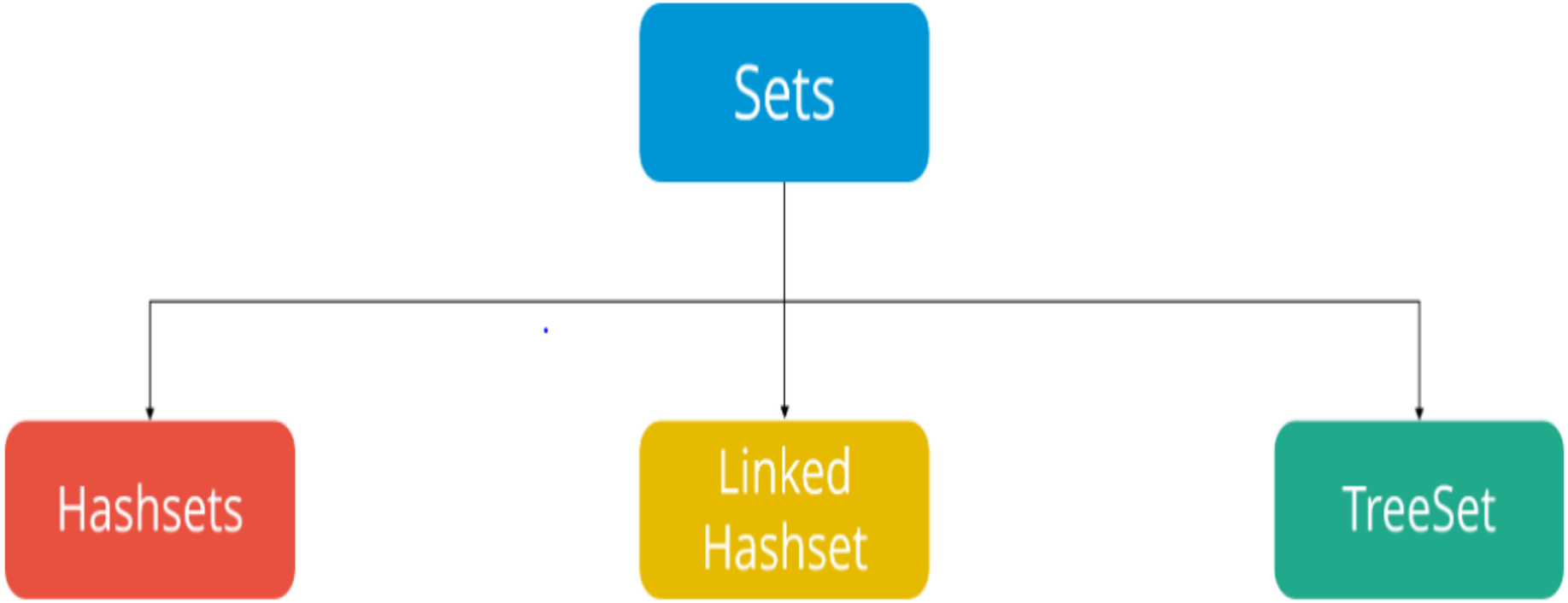
Queue Implementations

- **General-Purpose Queue Implementations**

  LinkedList implements the Queue interface, providing first in, first out (FIFO) queue operations for add, poll, and so on.
- The **PriorityQueue class** is a priority queue based on the heap data structure. This queue orders elements according to the order specified at construction time, which can be the elements' natural ordering, or the ordering imposed by an explicit Comparator.
- The queue retrieval operations — poll, remove, peek, and element — access the element at the head of the queue.
- The head of the queue is the least element with respect to the specified ordering.

Set Interface

- A Set refers to a **collection that cannot contain duplicate elements**. It is mainly used to model the mathematical set abstraction. Set has its implementation in various classes such as HashSet, TreeSet and LinkedHashSet.



Difference between HashSet, TreeSet and LinkedHashSet

	TreeSet	HashSet	LinkedHashSet
Duplicates	No	No	No
Thread safety	No	No	No
Speed	Slow	Faster	Medium
Order	Sorted order	No order	Insertion order
Null values	No	Allow	Allow
Implementation	By NavigableMap	HashMap	LinkedList & HashSet

HashSet Class

- Java **HashSet class** is used to create a collection that uses a **hash table for storage.** It inherits the AbstractSet class and implements Set interface.
- The important points about Java HashSet class are:
- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet **contains unique elements only.**
- HashSet **allows null value.**
- HashSet class is **non synchronized.**
- HashSet **doesn't maintain the insertion order.**
- Here, elements are **inserted on the basis of their hashcode.**
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

LinkedHashSet Class

Java **LinkedHashSet** class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

- Java LinkedHashSet class **contains unique elements** only like HashSet.
- Java LinkedHashSet class provides all optional set operation and permits null elements.
- Java LinkedHashSet class is **non synchronized.**
- Java LinkedHashSet class **maintains insertion order.**

TreeSet Class

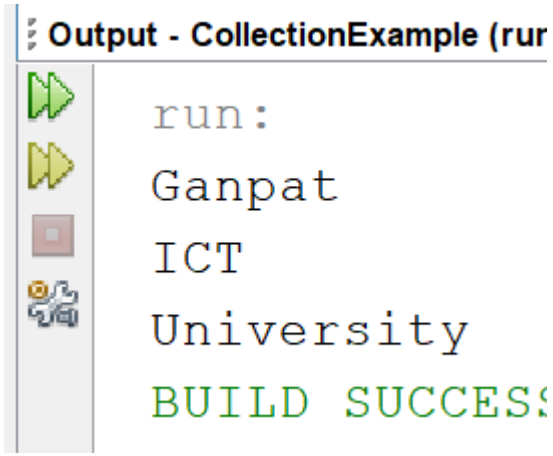
Java **TreeSet class implements the Set interface** that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

- Java TreeSet class **contains unique elements** only like HashSet.
- Java TreeSet class access and retrieval times are **quiet fast.**
- Java TreeSet class **doesn't allow null** element.
- Java TreeSet class is **non synchronized.**
- Java TreeSet class **maintains ascending order.**

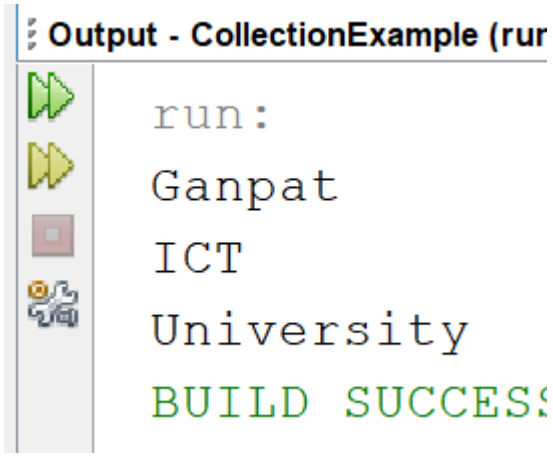
HashSet Class

```
package collectionexample;

import java.util.*;
class HashsetExample{
        public static void main(String args[])
        {
            HashSet<String> al=new HashSet();
            al.add("Ganpat");
            al.add("ICT");
            al.add("University");
            Iterator<String> itr=al.iterator();
            while(itr.hasNext())
            {
                System.out.println(itr.next());
            }
        }
}
```



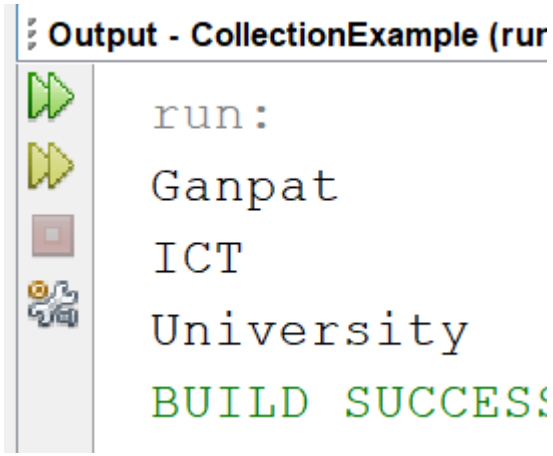
Linked HashSet Class

```
import java.util.*;
class LinkedHashsetExample{
    public static void main(String args[])
    {
        LinkedHashSet<String> al=new LinkedHashSet();
        al.add("Ganpat");
        al.add("ICT");
        al.add("University");
        Iterator<String> itr=al.iterator();
        while(itr.hasNext())
        {
            System.out.println(itr.next());
        }
    }
}
```



TreeSet Class

```
package collectionexample;

import java.util.*;
class TreesetExample{
        public static void main(String args[])
        {
            TreeSet<String> al=new TreeSet<String>();
            al.add("Ganpat");
            al.add("ICT");
            al.add("University");
            Iterator<String> itr=al.iterator();
            while(itr.hasNext())
            {
                System.out.println(itr.next());
            }
        }
}
```



Hashtable

```
103 CN
102 AEM
101 DBMS
100 OOP
```

```
package collectionexample;

import java.util.*;

public class HashTableDemo {
public static void main(String args[]){
        Hashtable<Integer,String> hm=new Hashtable<Integer,String>();
        hm.put(100,"OOP");
        hm.put(102,"AEM");
        hm.put(101,"DBMS");
        hm.put(103,"CN");
        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Comparator Interface

- Found in **java.util package** and contains 2 methods **compare(Object obj1,Object obj2) and equals(Object element).**

It provides **multiple sorting sequences**, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Comparator Interface

```
class Student
{
  int rollno;
  String name;
  int age;
  Student(int rollno,String name,int age)
  {
          this.rollno=rollno;
          this.name=name;
          this.age=age;
  }
}
```

```
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;
        }
}
```

Comparator Interface

```
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 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);
}
}
}
```

```
Sorting by age
    105 Jai 21
    101 Vijay 23
    106 Ajay 27
```