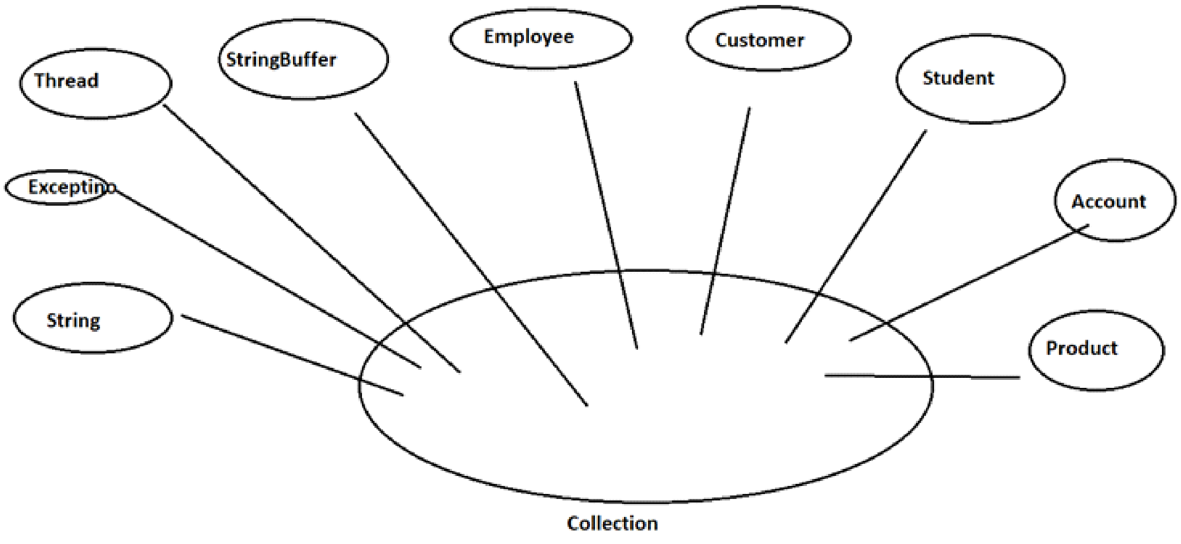
Collections

Collection is an object, it able to represent a group of other objects.



Q) <u>In Java Applications, To Represent A Group Of Other Elements We Have Already Arrays Then What Is The Requirement To Use Collections?</u>

OR

Q) <u>What Are The Differences Between Array And Collection?</u>

- **Arrays are having fixed size in nature. In case of arrays, we are able to add the elements up to the specified size only, we are unable to add the elements over its size, if we are trying to add elements over its size then JVM will rise an exception like "java.lang.ArrayIndexOutOfBoundsException".**

<u>EX:</u>

```
Student[] std=new Student[3];
std[0]=new Student();
std[1]=new Student();
std[2]=new Student();
std[3]=new Student();--> ArrayIndexOutOfBoundsException
```

Collections are having dynamically growable nature, even if we add the elements over its size then JVM will not rise any exception.

<u>EX:</u>

```
ArrayList al=new ArrayList(3);
al.add(new Student());
al.add(new Student());
al.add(new Student());
al.add(new Student());--> No Exception
```

- **In Java, by default, Arrays are able to allow homogeneous elements, if we are trying to add the elements which are not same Array data type then Compiler will rise an error like "Incompatible Types".**

EX:

```
Student[] std=new Student[3];
std[0]=new Student();
std[1]=new Student();
std[2]=new Customer();--> Incompatible Types Error
```

In Java, by default, Collections are able to allow heterogeneous elements, even we add different types of elements Compiler will not rise any error.

EX:

```
ArrayList al=new ArrayList(3);
al.add(new Student());
al.add(new Employee());---> No Error
al.add(new Customer();----> No Error
```

- Arrays are not having predefined methods to perform searching and sorting operations over the elements, in case of arrays to perform searching and sorting operations developers have to provide their own logic.

In case of Collections, predefined methods or predefined Collections are defined to perform Searching and sorting operations over the elements.

EX: In Collections, TreeSet was provided to perform sorting order.

```
TreeSet ts=new TreeSet();
ts.add("B");
ts.add("E");
ts.add("A");
ts.add("D");
ts.add("C");
ts.add("F");
System.out.println(ts);
```

OUTPUT: [A,B,C,D,E,F]

- Arrays are able to allow only one type of elements, so Arrays are able to improve Typedness in java applications and they are able to perform Type safe operations.

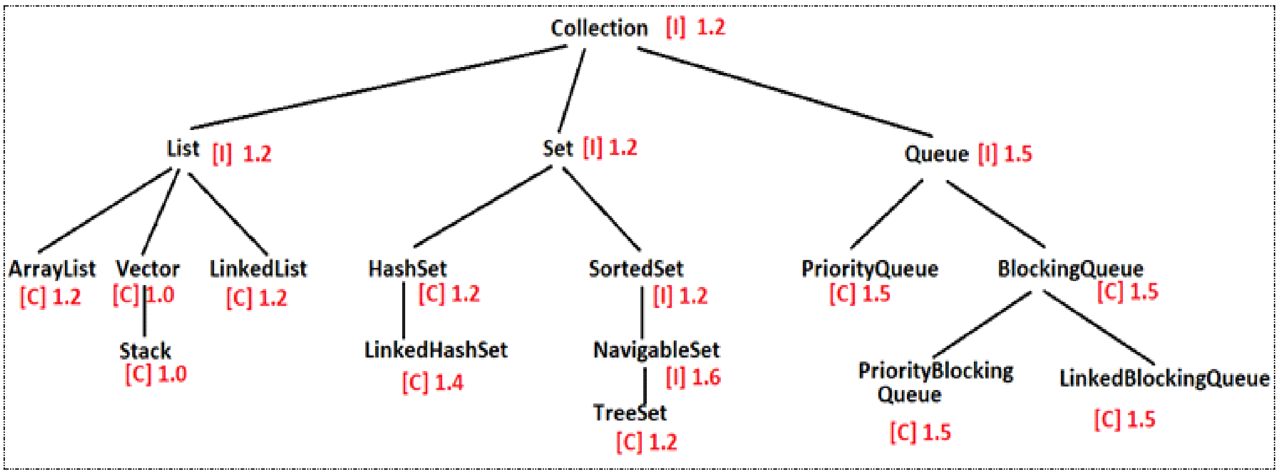
  Collections are able to allow different types of elements, so Collections are able to reduce typedness in java applications and they are unable to perform Type safe operations.

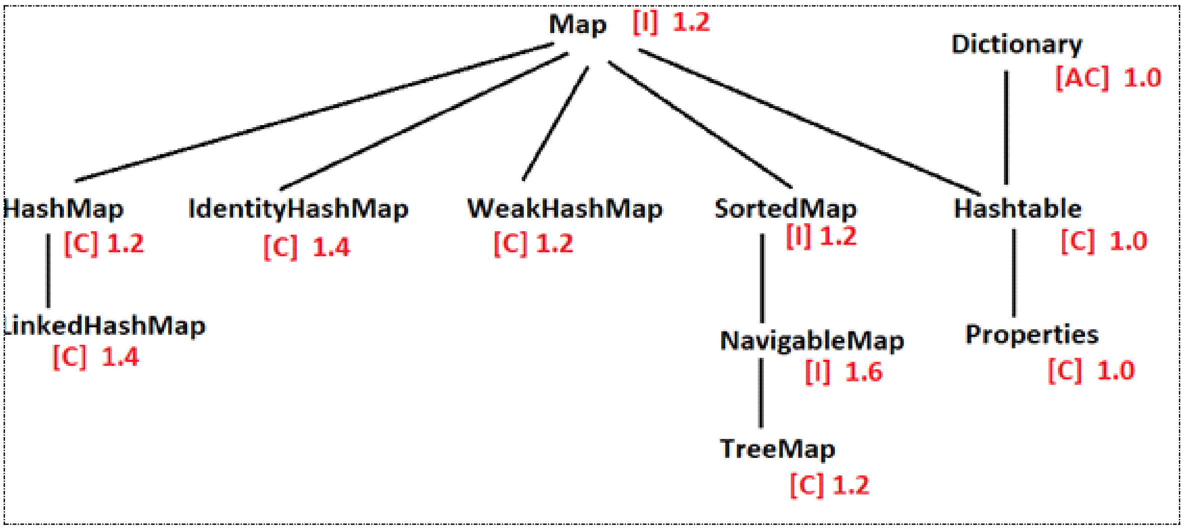
- If we know the no of elements in advance at the time of writing java applications then Arrays are better to use in java applications and they will provide very good performance in java applications, but, But Arrays are not flexible to design applications.

In java applications, Collections are able to provide less performance, but, they will provide flexibility to design applications.

To represent Collection objects in java applications, JAVA has provided predefined classes and interfaces in the form of java.util package called as "Collection Framework".

Q) What are the classes and interfaces are existed in java.util package to represent Collections?





Q) What are the differences between Collection and Map?

Collections are able to store all the elements individually, not in the form of Key-value pairs.

EX: To store 10 Employee objects we will use Collection.

Maps are able to store all the elements in the form of Key-value pairs.

EX: To represent Telephone Directory, where we are representing phone number and Customer Name we have to use Maps.

Q) What are the differences between List and Set?

- List is index based, it able to allow all the elements as per indexing.
  Set is not index based, it able to allow all the elements on the basis of elements hash code values.
- List is able to allow duplicate elements.
  Set is not allowing duplicate elements.
- List is able to allow any number of null values.
  Set is able to allow only one null value.
- List is following insertion order.
  Set is not following insertion order by default.
  Note: LinkedHashSet is following insertion order.

- List is not following sorting order.
  Sets are not following sorting order by default.
  Note: SortedSet, NavigableSet and TreeSet are following Sorting order.

- List is able to allow heterogeneous elements.
  Sets are able to allow heterogeneous elements by default.
  Note: SortedSett, NavigableSet and TreeSet are allowing only Homogeneous elements.

Collection:

- It is an interface provided by JAVA along with JDK 1.2 version.
- It able to represent a group of individual elements as single unit.
- It has provided the following methods common to every implementation class.

• public boolean add(Object obj)

This method is able to add the specified element to Collection object. If the specified element is added successfully then add(-) method will return "true" value. If the specified element is not added successfully then add() method will return "false" value.

EX:

```
import java.util.*;
class Test
{
   public static void main(String[] args)
```

```
    {
        HashSet hs=new HashSet();
        System.out.println(hs.add("A"));
        hs.add("B");
        hs.add("C");
        hs.add("D");
        System.out.println(hs);
        System.out.println(hs.add("A"));
        System.out.println(hs);
    }
}
```

OUTPUT:

```
true
[A,B,C,D]
false
[A,B,C,D]
```

- **public boolean addAll(Collection c)**

This method can be used to add all the elements of the specified Collection to the present Collection object. If addition operation is success then addAll(-) method will return "true" value, if addition operation is failure then addAll() method will return "false" value.

EX:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashSet hs=new HashSet();
        hs.add("A");
        hs.add("B");
        hs.add("C");
        hs.add("D");
        System.out.println(hs);
        HashSet hs1=new HashSet();
        System.out.println(hs1.addAll(hs));
        System.out.println(hs1);
        System.out.println(hs1.addAll(hs));
        System.out.println(hs1);
    }
}
```

OUTPUT:

```
[D,A,B,C]
true
[D,A,B,C]
false
[D,A,B,C]
```

• public boolean remove(Object obj)

This method can be used to remove the specified element from the Collection object. If remove operation is success then remove() method will return true value, if remove operation is failure then remove() method will return false value.

EX:

```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    System.out.println(al);
    System.out.println(al.remove("B"));
    System.out.println(al);
    System.out.println(al.remove("B"));
    System.out.println(al);
  }
}
```

OUTPUT:

```
[A,B,C,D]
true
[A,C,D]
false
[A,C,D]
```

• public boolean removeAll(Collection c)

This method can be used to remove all the elements of the specified Collection from the present Collection object. If remove operation is success then removeAll() method will return true value. If remove operation is not success then removeAll() method will return false value.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    al.add("F");
    System.out.println(al);
    ArrayList al1=new ArrayList();
    al1.add("B");
    al1.add("C");
    al1.add("D");
    System.out.println(al1);
    System.out.println(al.removeAll(al1));
    System.out.println(al);
    System.out.println(al.removeAll(al1));
    System.out.println(al);
  }
}
```

OUTPUT:

```
[A,B,C,D,E,F]
[B,C,D]
true
[A,E,F]
false
[A,E,F]
```

- **public boolean contains(Object obj)**

  This method will check whether the specified element is existed or not in the Collection object. If the specified element is existed then this method will return "true" value. If the specified element is not existed then this method will return "false" value.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    al.add("F");
    System.out.println(al);
    System.out.println(al.contains("B"));
    System.out.println(al.contains("X"));
  }
}
```

OUTPUT:

```
[A,B,C,D,E,F]
true
false
```

- ## public boolean containsAll(Collection c)

  This method will check whether all the elements of the specified Collection are available or not in the present Collection object. If all the elements are existed then containsAll() method will return true value, if atleast one element is not existed then containsAll() method will return false value.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    al.add("F");
    System.out.println(al);
    ArrayList al1=new ArrayList();
```

```
        al1.add("B");
        al1.add("C");
        al1.add("D");
        System.out.println(al.containsAll(al1));
        al1.add("X");
        al1.add("Y");
        System.out.println(al.containsAll(al1));
    }
}
```

<u>OUTPUT:</u>

```
[A,B,C,D,E,F]
true
false
```

- **<u>public boolean retainAll(Collection c)</u>**

  This method will remove all the elements from the present Collection object except the elements which are existed in the specified Collection object. If at least one element is removed then retainAll() method will return true value. If no elements are removed then retainsAll() method will return false value.

<u>EX:</u>

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList al=new ArrayList();
        al.add("A");
        al.add("B");
        al.add("C");
        al.add("D");
        al.add("E");
        al.add("F");
        System.out.println(al);
        ArrayList al1=new ArrayList();
        al1.add("B");
        al1.add("C");
        al1.add("D");
        System.out.println(al1);
        System.out.println(al.retainAll(al1));
        System.out.println(al);
        System.out.println(al.retainAll(al1));
        System.out.println(al);
    }
```

```
}
```

OUTPUT:

```
[A,B,C,D,E,F]
[B,C,D]
true
[B,C,D]
false
[B,C,D
```

- **public int size()**

  This method can be used to return an integer value representing the no of elements which are existed in the Collection object.

- **public void clear()**

  This method can be used to remove all elements from Collection object.

- **public boolean isEmpty()**

  This method can be used to check whether Collection object is empty or not. If the Collection object is empty then isEmpty() method will return "true" value. If the Collection object is not empty then isEmpty() method will return "false" value.

- **public Object[] toArray()**

  This method will return all the elements of the Collection object in the form of Object[].

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    al.add("F");
    System.out.println(al);
    System.out.println(al.size());
    Object[] obj=al.toArray();
    for(Object o: obj)
```

```
        {
            System.out.print(o+"  ");
        }
        System.out.println();
        System.out.println(al.isEmpty());
        al.clear();
        System.out.println(al.isEmpty());
        System.out.println(al);
    }
}
```

OUTPUT:

```
[A,B,C,D,E,F]
6
A B C D E F
false
true
[]
```

List:

- List is a direct child interface to Collection interface
- List was provided by JAVA along with its JDK1.2 version
- List is index based, it able to arrange all the elements as per indexing.
- List is able to allow duplicate elements.
- List is following insertion order.
- List is not following Sorting order.
- List is able to allow any number of null values.
- List is able to allow heterogeneous elements.

List interface has provided the following methods common to all of its implementation classes.

- **public void add(int index, Object obj)**

  It able to add the specified element at the specified index value.

- **public Object set(int index, Object obj)**

  It able to set the specified element at the specified index value.

Q) What is the difference between *add(--) Method* and *set(--) Method?*

- add(--) method is able to perform insert operation. If any element is existed at the specified element then add() method will insert the specified new element at the specified index value and add() method will adjust the existed element to next index value. If no element is existed at the specified index then add() method add the specified element at the specified index.

- set(--) method is able to perform replace operation. If any element is existed at the specified index then set() method will remove the existed element and set(-) method will add the specified element to the specified index and set() method will return the removed element. If no element is existed at the specified index value then set() method will rise an exception like java.lang.indexOutOfBoundsException.

- **public Object get(int index)**
  It will return an element available at the specified index value.

- **public Object remove(int index)**
  It will remove and return an element available at the specified index value.

- **public int indexOf(Object obj)**
  It will return an index value where the first occurrence of the specified element.

- **public int lastIndexOf(Object obj)**
  It will return an index value where the last occurrence of the specified element.

EX:

```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    System.out.println(al);
    al.add(1,"X");
    System.out.println(al);
    al.add(6,"F");
    System.out.println(al);
    al.set(3,"Y");
    System.out.println(al);
    //al.set(7,"Z");--->IndexOutOfBoundsException
    System.out.println(al.get(4));
    System.out.println(al.remove(6));
```

```
        System.out.println(al);
        al.add(6,"X");
        al.add(7,"B");
        al.add(8,"X");
        System.out.println(al);
        System.out.println(al.indexOf("X"));
        System.out.println(al.lastIndexOf("X"));
    }
}
```

ArrayList:

- It was provided by JAVA along with JDK 1.2 version.
- It is a direct implementation class to List interface.
- It is index based.
- It allows duplicate elements.
- It follows insertion order.
- It will not follow sorting order.
- It allows heterogeneous elements.
- It allows any number of null values.
- Its internal data structure is "Resizable Array".
- Its initial capacity is 10 elements.
- Its incremental capacity ration is
  new_Capacity = (Current_Capacity*3/2)+1
- It is best option for frequent retrieval operations.
- It is not synchronized.
- No method is synchronized method in ArrayList.
- It allows more than one thread to access data.
- It follows parallel execution.
- It will reduce execution time.
- It will improve application performance.
- It will not give guarantee for data consistency.
- It is not thread safe.
- It is not Legacy Collection.

Constructors:

- **public ArrayList()**
  It can be used to create an empty ArrayList object with 10 elements as default capacity value.

EX: ArrayList al = new ArrayList();

- **public ArrayList(int capacity)**

  It can be used to create an empty ArrayList object with the specified capacity.

  EX: ArrayList al = new ArrayList(20);

- **public ArrayList(Collection c)**

  It can be used to create an ArrayList object with all the elements of the specified Collection object.

EX:

```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al1=new ArrayList();
    al1.add("AAA");
    al1.add("BBB");
    al1.add("CCC");
    al1.add("DDD");
    System.out.println(al1);
    ArrayList al2=new ArrayList(al1);
    System.out.println(al2);
  }
}
```

OUTPUT:

```
[AAA,BBB,CCC,DDD]
[AAA,BBB,CCC,DDD]
```

EX:

```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
```

```
        ArrayList al=new ArrayList();
        al.add("A");
        al.add("B");
        al.add("C");
        al.add("D");
        al.add("E");
        System.out.println(al);
        al.add("B");
        System.out.println(al);
        al.add(new Integer(10));
        System.out.println(al);
        al.add(null);
        al.add(null);
        System.out.println(al);
    }
}
```

Vector:

- It was introduced in JDK1.0 version.
- It is Legacy Collection.
- It is a direct implementation class to List interface.
- It is index based.
- It allows duplicate elements.
- It follows insertion order.
- It will not follow sorting order.
- It allows heterogeneous elements.
- It allows any number of null values.
- Its internal data structure is "Resizable Array".
- Its initial capacity is 10 elements.
- It is best choice for frequent retrieval operations.
- It is not good for frequent insertions and deletion operations.
- Its incremental capacity is double the current capacity.
  New_capacity = 2*Current_Capacity
- It is synchronized elemenet.
- All the methods of vector class are synchronized.
- It allows only one thread at a time.
- It follows sequential execution.
- It will increase execution time.
- It will reduce application performance.
- It is giving guarantee for data consistency.
- It is threadsafe.

Constructors:

- **<u>public Vector()</u>**

  It can be used to create an empty Vector object with the initial capacity 10 elements.

  <u>EX:</u>
  ```
  Vector v = new Vector();
  System.out.println(v.capacity());
  ```
  <u>OUTPUT:</u> 10

- **<u>public Vector(int capacity)</u>**

  It can be used to create an empty vector object with the specified capacity value.

  <u>EX:</u>
  ```
  Vector v = new Vector(20);
  System.out.println(v.capacity());
  ```
  <u>OUTPUT:</u> 20

- **public Vector(int capacity, int incremental_Ratio)**

This constructor can be used to create an empty Vector object with the specified initial capacity and with the specified incremental ratio.

<u>EX:</u>

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    Vector v=new Vector(5,5);
    System.out.println(v.capacity());
    for(int i=1;i<=6;i++)
    {
      v.add(i);
    }
    System.out.println(v.capacity());
    for(int i=7;i<=11;i++)
    {
      v.add(i);
    }
    System.out.println(v.capacity());
  }
}
```

<u>OUTPUT:</u>

```
5
10
15
```

- **public Vector(Collection c)**
  This constructor can be used to create Vector object with all the elements of the specified Collection object.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    Vector v=new Vector();
    v.add("A");
    v.add("B");
    v.add("C");
    v.add("D");
    System.out.println(v);
    Vector v1=new Vector(v);
    System.out.println(v1);
  }
}
```

OUTPUT:

```
 [A,B,C,D]
[A,B,C,D]
```

Methods:

- **public void addElement(Object obj)**
  It will add the specified element to Vector.

- **public Object firstElement()**
  It will return first element of the Vector.

- **public Object lastElement()**
  It will return last element of the Vector.

- **public Object elementAt(int index)**
  It will return an element available at the specified index.

- **public void removeElement(Object obj)**

It will remove the specified element from Vector.

- **public void removeElementAt(int index)**

  It will remove an element existed at the specified index value.

- **public void removeAllElements()**

  It will remove all elements from Vector.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    Vector v=new Vector();
    v.addElement("A");
    v.addElement("B");
    v.addElement("C");
    v.addElement("D");
    v.addElement("E");
    System.out.println(v);
    System.out.println(v.firstElement());
    System.out.println(v.lastElement());
    System.out.println(v.elementAt(3));
    v.removeElement("D");
    System.out.println(v);
    v.removeElementAt(2);
    System.out.println(v);
    v.removeAllElements();
    System.out.println(v);
  }
}
```

Q) What are the differences between *ArrayList* and *Vector?*

- **ArrayList Class was introduced in JDK 1.2 Version**

  Vector class was introduced in JDK1.0 version.

- **ArrayList is not Legacy Collection**
  Vector is Legacy Collection.

- **ArrayList is not synchronized**
  Vector is synchronized.

- **No method is synchronized Method in ArrayList**
  Almost all the methods are synchronized methods in vector.

- **ArrayList allows more than one Thread at a Time to access Data**
  Vector allows only one thread at a time to access data.

- **ArrayList follows parallel Execution**
  Vector follows sequential execution.

- **ArrayList is able to reduce Application Execution Time**
  Vector is able to increase application execution time.

- **ArrayList is able to improve Application Performance**
  Vector is able to reduce application performance.

- **ArrayList is not giving guarantee for data consistency.**
  Vector is giving guarantee for data consistency.

- **ArrayList is not threadsafe.**
  Vector is threadsafe.

- **ArrayList Incremental Capacity is (Current_Capacity*3/2)+1**
  Vector incremental capacity is 2*Current_Capacity

- **We are unable to get Capacity Value of ArrayList, because, no capacity() Method in ArrayList Class.**
  We can get capacity value of Vector, because, capacity() method is existed in vector class.

Stack:

It was introduced in JDK 1.0 version, it is a Legacy Collection and it is a child class to Vector class. It able to arrange all the elements as per "Last In First Out" [LIFO] alg.

Constructor:

public Stack()
It will create an empty Stack object.
EX: Stack s = new Stack();

Methods:

- **public void push(Object obj)**
  It will add the specified element to Stack.

- **public Object pop()**
  It will remove and return top of the stack.

- **public Object peek()**
  It will return top of the stack.

- **public int search(Object obj)**
  It will check whether the specified element is existed or not in the Stack, if the specified element is not existed then it will return '-1' value, if the specified element is existed then it will return its position.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    Stack s=new Stack();
    s.push("A");
    s.push("B");
    s.push("C");
    s.push("D");
    s.push("E");
    System.out.println(s);
    System.out.println(s.pop());
    System.out.println(s);
    System.out.println(s.peek());
    System.out.println(s);
    System.out.println(s.search("B"));
    System.out.println(s.search("X"));
  }
```

20) }

LinkedList:

- It was introduced in JDK1.2 version.
- It is not Legacy Collection.
- It is a direct implementation class to List interface.
- It is index based.
- It allows duplicate elements.
- It follows insertion order.
- It is not following sorting order.
- It allows heterogeneous elements.
- It allows null values in any number.
- Its internal data structure is "Double Linked List".;
- It is best choice for frequent insertions and deletions.
- It is not synchronized Collection.
- No method is synchronized in LinkedList.
- It allows more than one thread to access data.
- It will follow parallel execution.
- It will decrease execution time.
- It will improve application performance.
- It is not giving guarantee for data consistency.
- It is not threadsafe.

Constructors:

- **public LinkedList()**
  It will create an empty LinkedList object.
  EX: LinkedList ll = new LinkedList();

- **public LinkedList(Collection c)**
  It will create LinkedList object with all the elements of the specified Collection object.

EX:
```
LinkedList ll1 = new LinkedList();
ll1.add("A");
ll1.add("B");
ll1.add("C");
ll1.add("D");
System.out.println(ll1);
LinkedList ll2=new LinkedList(ll1);
System.out.println(ll2);
```

OUTPUT:
```
[A, B, C, D]
[A, B, C, D]
```

Methods:

- **public void addFirst(Object obj)**
  It will add the specified element as first element to LinkedList.

- **public void addLast(Object obj)**
  It will add the specified element as last element to LinkedList.

- **public Object getFirst()**
  It will return first element from LinkedList.

- **public Object getLast()**
  It will return last element from LinkedList.

- **public void removeFirst()**
  It will remove first element from LinkedList.

- **public void removeLast()**
  It will remove last element from LinkedList.

EX:
```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    LinkedList ll=new LinkedList();
    ll.add("A");
    ll.add("B");
    ll.add("C");
    ll.add("D");
    ll.add("E");
    System.out.println(ll);
    ll.addFirst("X");
    ll.addLast("Y");
    System.out.println(ll);
    ll.removeFirst();
    ll.removeLast();
    System.out.println(ll);
```

```
        System.out.println(ll.getFirst());
        System.out.println(ll.getLast());
    }
}
```

Cursors / Iterators in Collections:

In java applications, when we pass Collection object reference variable as parameter to System.out.println(-) method, then, JVM will execute toString() method internally. Initially toString() method was implemented in java.lang.Object class, it was implemented in such a way that to return a String contains "Class_Name@Ref_val" . In java applications, Collection classes are not depending on Object class toString() method, they are having their own toString() method , which are implemented in such a way to return a String contains all the elements of the Collection object by enclosed with [].

EX:

```
ArrayList al = new ArrayList();
al.add("A");
al.add("B");
al.add("C");
al.add("D");
System.out.println(al);
```

OUTPUT: [A, B, C, D]

As per the requirement, we don’t want to display all the Elements at a time on command prompt, we want to retrieve elements one by one individually from Collection objects and we want to display all the elements one by one on Command prompt.

To achieve the above requirement, Collection Framework has provided the following three Cursors or Iterators.

- **Enumeration**
- **Iterator**
- **ListIterator**

• Enumeration:

- **It is a Legacy Cursor, it is applicable for only Legacy Collections to retrieve elements in one by one fashion.**

- To retrieve elements from Collections by using Enumeration we have to use the following steps.

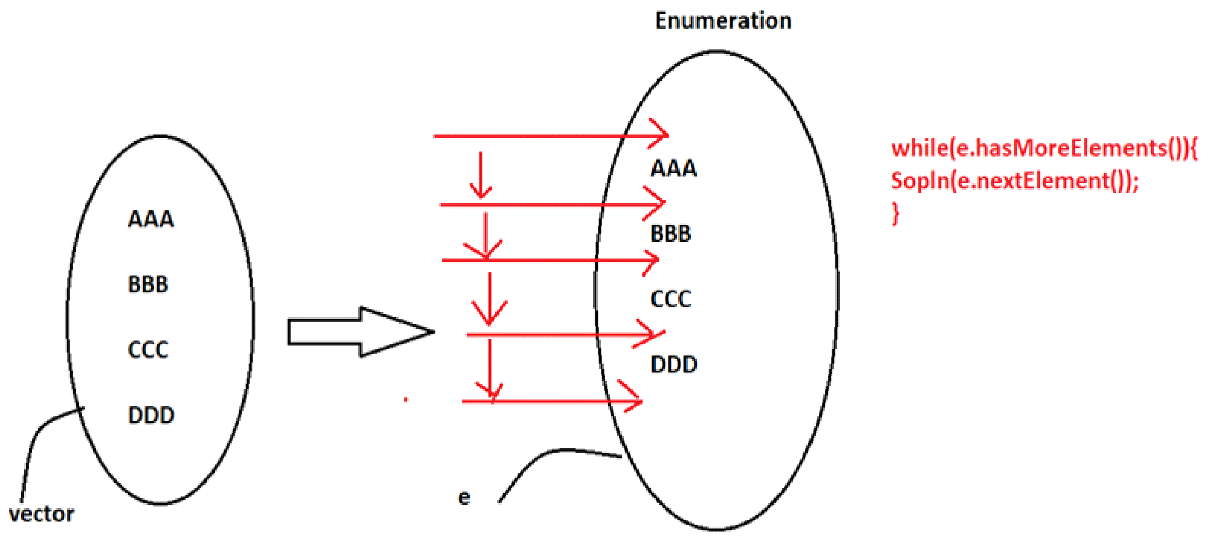
- **Create Enumeration Object:**

  To create Enumeration object we have to use the following method from Legacy Collections.
  public Enumeration elements()

- **Retrive Elements from Enumeration:**

  - Check whether more elements are available or not from Current cursor position by using the following method.
    public boolean hasMoreElements()
    - It will return true value if at least next element is existed.
    - It will return false value if no element is existed from current cursor position.

  - If at least next element is existed then read next element and move cursor to next position by using the following method.
    public Object nextElement()



EX:

```
import java.util.*;
class Test
{
```

```
    public static void main(String[] args)
    {
        Vector v=new Vector();
        v.add("A");
        v.add("B");
        v.add("C");
        v.add("D");
        v.add("E");
        System.out.println(v);
        Enumeration e=v.elements();
        while(e.hasMoreElements())
        {
        System.out.println(e.nextElement());
        }
    }
}
```

Drawbacks:

- Enumeration is applicable for only Legacy Collections.
- Enumeration is able to allow only read operation while iterating elements.

• Iterator:

- Iterator is an interface provided JAVA along with its JDK1.2 version.
- Iterator can be used to retrieve all the elements from Collection objects in one by one fashion.
- Iterator is applicable for all the Collection interface implementation classes to retrieve elements.
- Iterator is able to allow both read and remove operations while iterating elements.
- If we want to use Iterator in java applications then we have to use the following steps.

• Create Iterator Object:

- To create Iterator object we have to use the following method from all Collection implementation classes.
- public Iterator iterator()
- **EX:** Iterator it = al.iterator();

• Retrive Elements from Iterator:

- To retrieve elements from Iterator we have to use the following steps.
- Check whether next element is existed or not from the current cursor position by using the following method.

public boolean hasNext()

- This method will return true if next element is existed.
- This method will return false if no element is existed from current cursor position.

- If next element is existed then read next element and move cursor to next position by using the following method.
- public Object next()

NOTE: To remove an element available at current cursor position then we have to use the following method
public void remove()

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("A");
    al.add("B");
    al.add("C");
    al.add("D");
    al.add("E");
    System.out.println(al);
    Iterator it=al.iterator();
    while(it.hasNext())
    {
      String element=(String)it.next();
      System.out.println(element);
      if(element.equals("C"))
      {
        it.remove();
      }
    }
    System.out.println(al);
  }
}
```

Q) What are the differences between *Enumeration* and *Iterator?*

- Enumeration is Legacy Cursor, it was introduced in JDK1.0 version.
  Iterator is not Legacy Cursor, it was introduced in JDK1.2 version.

- Enumeration is not Universal Cursor, it is applicable for only Legacy Collections.

Iterator is an universal Cursor, it is applicable for all Collection implementations.

- Enumeration is able to allow only read operation while iterating elements.
  Iterator is able to allow both read operation and remove operation while iterating elements.

ListIterator:

- It is an interface provided by JAVA along with JDK1.2 version.
- It able to allow to read elements in both forward direction and backward direction.
- It able to allow the operations like read, insert, replace and remove while iterating elements.
- If we want to use ListIterator in java applications then we have to use the following steps.

Create ListIterator Object:

- To create ListIterator object we have to use the following method.
- public ListIterator listIterator()
- **EX:** ListIterator lit = ll.listIterator();

Retrieve Elements from ListIterator

To retrieve elements from ListIterator in Forward direction then we have to use the following methods.

public boolean hasNext()

It will check whether next element is existed or not from the current cursor position.

public Object next()

It will return next element and it will move cursor to the next position in forward direction.

public int nextIndex()

It will return next index value from the current cursor position.

To retrieve elements in Backward direction we have to use the following methods.

public boolean hasPrevious()

It will check whether previous element is existed or not from the current cursor position, If previous element is existed then it will return "true" value, if previous element is not existed then it will return false value.

public Object previous()

It will return previous element and it will move cursor to the next previous position.

public int previousIndex()

It will return previous index value from the current cursor position.

To perform the operations like remove, insert and replace over the elements we have to use the following methods.

- public void remove()
- public void add(Object obj)
- public void set(Object obj)

EX:

```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    LinkedList ll=new LinkedList();
    ll.add("A");
    ll.add("B");
    ll.add("C");
    ll.add("D");
    ll.add("E");
    ll.add("F");
    System.out.println(ll);
    ListIterator lit=ll.listIterator();
    System.out.println("Elements in Forward Direction");
    while(lit.hasNext())
    {
      System.out.println(lit.nextIndex()+"--->"+lit.next());
    }
    System.out.println();
    System.out.println("Elements in Backward Direction");
    while(lit.hasPrevious())
    {
      System.out.println(lit.previousIndex()+"--->"+lit.previous());
    }
  }
}
```

EX:

```
import java.util.*;
class Test1
{
  public static void main(String[] args)
  {
    LinkedList ll=new LinkedList();
    ll.add("A");
    ll.add("B");
    ll.add("C");
    ll.add("D");
    ll.add("E");
    ll.add("F");
    System.out.println(ll);
    ListIterator lit=ll.listIterator();
    while(lit.hasNext())
    {
      String element=(String)lit.next();
      if(element.equals("B"))
      {
        lit.add("X");
      }
      if(element.equals("D"))
      {
        lit.set("Y");
      }
      if(element.equals("E"))
      {
        lit.remove();
      }
    }
    System.out.println(ll);
  }
}
```

Q) <u>What are the differences between *Enumeration, Iterator* and *ListIterator?*</u>

- Enumeration is applicable for only Legacy Collections.
  Iterator is applicable for all Collection implementations.
  ListIterator is applicable for only List implementations.

- Enumeration and Iterator are allowed to iterate elements in only Forward direction.
  ListIterator is able to allow to iterate elements in both Forward direction and backward direction.

- Enumeration is able to allow only read operation while iterating elements.
  Iterator is able to allow both read and remove operations while iterating elements.
  ListIterator is able to allow the operations like insert, replace, remove and read operations while iterating elements.

Set:

- It was introduced in JDK 1.2 version.
- It is a direct child interface to Collection interface.
- It is not index based, it able to arrange all the elements on the basis of elements hashcode values.
- It will not allow duplicate elements.
- It will not follow insertion order.
  Note: LinkedHashSet will follow insertion order.
- It will not follow Sorting order.
  Note: SortedSet, NavigableSet and TreeSet are following Sorting order.
- It able to allow only one null value.
  Note: SortedSet, NavigableSet and TreeSet are not allowing even single null value.

HashSet:

- HashSet is a direct implementation class to Set interface.
- It was introduced in JDK 1.2 version.
- It is not index based, it able to arrange all the elements on the basis of elements hashcode values.
- It will not allow duplicate elements.
- It will not follow insertion order.
- It will not follow Sorting order.
- It able to allow only one null value.
- Its interal data structer is "Hashtable".
- Its initial capacity is "16" elements and its initial fill_Ratio is 75%.
- It is not synchronized.
- Almost all the methods are not synchronized in HashSet
- It allows more than one thread at a time.
- It follows parallel execution.
- It will reduce execution time.

- It improves performance of the applications.
- It is not giving guarantee for data consistency.
- It is not threadsafe.

Constuctors:

- **public HashSet()**

  This constructor can be used to create an empty HashSet object with 16 elements as initial capacity and 75% fill ratio.

  EX: HashSet hs = new HashSet();

- **public HashSet(int capacity)**

  This constructor can be used to create an empty HashSet object with the specified capacity as initial capacity and with the default fill ratio 75%.

  EX: HashSet hs = new HashSet(20);

- **public HashSet(int capacity, float fill_Ratio)**

  This constructor can be used to create an empty HashSet object with the specified capacity and with the specified fill ratio.

  EX: HashSet hs = new HashSet(20, 0.85f);

- **public HashSet(Collection c)**

  This constructor can be used to create HashSet object with all the elements of the specified Collection.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    HashSet hs1=new HashSet();
    hs1.add("A");
    hs1.add("B");
    hs1.add("C");
    hs1.add("D");
    hs1.add("E");
    System.out.println(hs1);
    HashSet hs2=new HashSet(hs1);
    System.out.println(hs2);
  }
}
```

OUTPUT:
```
[D,E,A,B,C]
[D,E,A,B,C]
```

EX:
```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    HashSet hs=new HashSet();
    hs.add("A");
    hs.add("B");
    hs.add("C");
    hs.add("D");
    hs.add("E");
    System.out.println(hs);
    hs.add("B");
    System.out.println(hs);
    hs.add(null);
    hs.add(null);
    System.out.println(hs);
    hs.add(new Integer(10));
    System.out.println(hs);
  }
}
```

LinkedHashSet:

Q) What are the differences between *HashSet* and *LinkedHashSet?*

- HashSet was introduced in JDK 1.2 version.
  LinkedhashSet was introduced in JDK 1.4 version.

- HashSet is not following insertion order.
  LinkedHashSet is following insertion order.

- The internal data structer of HashSet is "Hashtable".
  The internal data structer of LinkedHashSet is "Hashtable" and "LinkedList".

EX:
```java
import java.util.*;
```

```
class Test
{
  public static void main(String[] args)
  {
    HashSet hs=new HashSet();
    hs.add("A");
    hs.add("B");
    hs.add("C");
    hs.add("D");
    hs.add("E");
    System.out.println(hs);
      LinkedHashSet lhs=new LinkedHashSet();
    lhs.add("A");
    lhs.add("B");
    lhs.add("C");
    lhs.add("D");
    lhs.add("E");
    System.out.println(lhs);
  }
}
```

OUTPUT:

```
[D,E,A,B,C]
[A,B,C,D,E]
```

SortedSet:

- It was introduced in JDK1.2 version.
- It is a child interface to Set interface.
- It is not index based.
- It is not allowing duplicate elements.
- It is not following insertion order.
- It follows Sorting order.
- It allows only homogeneous elements.
- It will not allow heterogeneous elements, if we are trying to add heterogeneous elements then JVM will rise an exception like java.lang.ClasscastException.
- It will not allow null values, if we are trying to add any null value then JVM will rise an exception like java.lang.NullPointerException.
- It able to allow only Comparable objects by default, if we are trying to add non comparable objects then JVM will rise an exception like java.lang.ClassCastException.

Note: If we are trying to add non comparable objects then we have to use Comparator.

Methods:

- **public Object first()**

  It will return first element from SortedSet.

- **public Object last()**

  It will return last element from SortedSet.

- **public SortedSet headSet(Object obj)**

  It will return SortedSet object with the elements which are less the specified element.

- **public SortedSet tailSet(Object obj)**

  It will return SoredSet object with the elements which are greater than or equals to the specified element.

- **public SortedSet subSet(Object obj1, Object obj2)**

  It will return SortedSet object with all elements which are greater than or equals to the specified first element and which are less than the specified second element.

EX:

```
import java.util.*;
class Test
{
   public static void main(String[] args)
   {
      TreeSet ts=new TreeSet();
      ts.add("D");
      ts.add("F");
      ts.add("B");
      ts.add("E");
      ts.add("C");
      ts.add("A");
      System.out.println(ts);
      System.out.println(ts.first());
      System.out.println(ts.last());
      System.out.println(ts.headSet("D"));
      System.out.println(ts.tailSet("D"));
      System.out.println(ts.subSet("B","E"));
   }
}
```

NavigableSet

It was introduced in JAVA6 version, it is a child interface to SortedSet interface, it is following all the properties of SortedSet and it has define methods to provide navigations over the elements.

Methods:

- **public Object ceiling(Object obj)**
  It will return lowest element among all the elements which are greater than or equals to the specified element.
- **public Object higher(Object obj)**
  It will return lowest element among all the elements which are greater than the specified element.
- **public Object floor(Object obj)**
  It will return highest element among all the elements which are less than or equals to the specified element.
- **Trpublic Object lower(Object obj)**
  It will return highest element among all the elements which are less than the specified element.
- **public Object pollFirst()**
  It will remove and return first element from NavigableSet.
- **public Object pollLast()**
  It will remove and return last element from NavigableSet.
- **public NavigableSet descendingSet()**
  It will return all elements in the form of NavigableSet in descending order.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    TreeSet ts=new TreeSet();
    ts.add("D");
    ts.add("F");
    ts.add("B");
```

```
        ts.add("E");
        ts.add("C");
        ts.add("A");
        System.out.println(ts);
        System.out.println(ts.ceiling("D"));
        System.out.println(ts.higher("D"));
        System.out.println(ts.floor("D"));
        System.out.println(ts.lower("D"));
        System.out.println(ts.descendingSet());
        ts.pollFirst();
        ts.pollLast();
        System.out.println(ts);
    }
}
```

TreeSet:

- It was introduced in JDK 1.2 version.
- It is not Legacy Collection.
- It has provided implementation for Collection, Set, SortedSet and NavigableSet interfaces.
- It is not index based.
- It is not allowing duplicate elements.
- It is not following insertion order.
- It follows Sorting order.
- It allows only homogeneous elements.
- It will not allow heterogeneous elements, if we are trying to add heterogeneous elements then JVM   will rise an exception like java.lang.ClasscastException.
- It will not allow null values, if we are trying to add any null value then JVM will rise an exception like java.lang.NullPointerException.
- It able to allow only Comparable objects by default, if we are trying to add non comparable objects then JVM will rise an exception like java.lang.ClassCastException.

NOTE: If we are trying to add non comparable objects then we have to use java.util.Comparator.

- Its internal data structure is "Balanced Tree".
- It is mainly for frequent search operations.

Constructors:

- **public TreeSet()**

  It can be used to create an Empty TreeSet object.

  EX: TreeSet ts = new TreeSet();

- **public TreeSet(Comparator c)**

  It will create an empty TreeSet object with the explicit Sorting mechanism in the form of Comparator

  EX: TreeSet ts = new TreeSet(new MyComparator());

- **public TreeSet(SortedSet ts)**

  It will create TreeSet object with all elements of the specified SortedSet.

EX:

```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    TreeSet ts1=new TreeSet();
    ts1.add("B");
    ts1.add("C");
    ts1.add("F");
    ts1.add("A");
    ts1.add("E");
    ts1.add("D");
    System.out.println(ts1);
    TreeSet ts2=new TreeSet(ts1);
    System.out.println(ts2);
  }
}
```

4) **public TreeSet(Collection c)**

It able to create TreeSet object with all the elements of the specified Collection.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    ArrayList al=new ArrayList();
    al.add("B");
    al.add("C");
    al.add("F");
    al.add("A");
    al.add("E");
    al.add("D");
    System.out.println(al);
    TreeSet ts=new TreeSet(al);
    System.out.println(ts);
  }
}
```

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    TreeSet ts=new TreeSet();
    ts.add("B");
    ts.add("C");
    ts.add("F");
    ts.add("A");
    ts.add("E");
    ts.add("D");
    System.out.println(ts);
    ts.add("B");
    System.out.println(ts);
    //ts.add(null);--> NullPointerException
    //ts.add(new Integer(10));-->ClassCastException
    //ts.add(new StringBuffer("BBB"));->ClassCastException
  }
}
```

When we add elements to the TreeSet object, TreeSet object will arrange all the elements in a particular sorting order with the following algorithm.

- TreeSet will construct a Tree [Balanced Tree] on the basis of the elements.

To construct Balanced Tree we have to use the following steps.

- If the element is first element to the TreeSet object then make that element as "Root Node".
- If the element is not first element then access compareTo(--) method over the present element by passing previous elements one by one of the balanced Tree right from root node until the present element is located in Tree.
    - If compareTo(-) method returns -ve value then go to left child of the present node and access again compareTo(-) method by passing left child. If no left child is existed then make the present element as left child
    - If compareTo(-) method returns +ve value then go to right child and access again compareTo(-) by passing right as parameter. If no right child is existed then make the present element as right child.
    - If compareTo(-) method return 0 value then discard the present element and declare that the present element is a duplicate element of the existed element.

- TreeSet will retrieve all the elements from balanced Tree by following in order traversal.
  In String class, compareTo(-) method was implemented like below
  str1.compareTo(str2);

    - If str1 come first when compared with str2 as per dictionary order then compareTo() method will return -ve value.
    - If str2 come first when compared with str1 in dictionary order then compareTo() method will return +ve value.
    - If str1 and str2 are same or available at same location in dictionary order then compareTo(-) method will return 0 value.

If we want to add user defined elements like Employee, Student, Customer to TreeSet then we have to use the following steps.

- Declare an user defined class.
- Implement java.lang.Comparable iterface in User defined class.
- Provide implementation for compareTo(-) method in user defined class.
- In main class, in main() method, create objects for user defined class and add objects to TreeSet object.

```
package com.durgasoft.app05.test;

import java.util.TreeSet;

class Employee implements Comparable{
    int eno;
    String ename;
    float esal;
    String eaddr;

    public Employee(int eno, String ename, float esal, String eaddr){
        this.eno = eno;
        this.ename = ename;
        this.esal = esal;
        this.eaddr = eaddr;
    }

    @Override
    public int compareTo(Object obj) {
        Employee emp = (Employee)obj;
        int val = 0;
        val = this.ename.compareTo(emp.ename);
        return -val;
    }

    @Override
    public String toString() {
        return "\nEmployee{" +
                "eno=" + eno +
                ", ename='" + ename + '\'' +
                ", esal=" + esal +
                ", eaddr='" + eaddr + '\'' +
                '}';
    }
}
public class Test {
    public static void main(String[] args) {
        Employee emp1 = new Employee(111, "Durga", 5000, "Hyd");
        Employee emp2 = new Employee(222, "Neelesh", 6000, "Hyd");
        Employee emp3 = new Employee(333, "Aishwary", 7000, "Hyd");
        Employee emp4 = new Employee(444, "Gopi Krishna", 8000, "Hyd");
        Employee emp5 = new Employee(555, "Tahauddin", 9000, "Hyd");

        TreeSet ts = new TreeSet();
        ts.add(emp1);
        ts.add(emp2);
        ts.add(emp3);
        ts.add(emp4);
        ts.add(emp5);

```

```
        System.out.println(ts);
    }
}
```

If we add non-comparable objects to TreeSet object then JVM will rise an exception like java.lang.ClassCastException, because, Non-Comparable objects are not providing compareTo(-) method internally, but, it is required to the TreeSet inorder to provide sorting order over elements.

If we want to add non-Comparable objects to TreeSet object then we must provide sorting logic to TreeSet object explicitly, for this, we have to use java.util.Comparator interface.

If we want to use Comparator interface in java applications then we have to use the following steps.

- Declare an User defined class.
- Implement java.util.Comparator interface in user defined class.
- Provide implementation for Comparator interface methods in user defined class.

```
public boolean equals(Object obj)
public int compare(Object obj1, Object obj2)
```

Note: In User defined class it is not required to implement equals(-) method, because, equals(-) method will come from default super class Object.

- Provide User defined Comparator object to TreeSet object

EX:
```
MyComparator mc = new MyComparator();
TreeSet ts = new TreeSet(mc);
```

EX:
```
import java.util.*;
class MyComparator implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
    StringBuffer s1=(StringBuffer)obj1;
    StringBuffer s2=(StringBuffer)obj2;

    int length1=s1.length();
    int length2=s2.length();
    int val=0;
    if(length1<length2)
    {
      val=-100;
    }
    else if(length1>length2)
```

```
        {
            val=100;
        }
        else
        {
            val=0;
        }
        return -val;
    }
}
class Test
{
    public static void main(String[] args)
    {
        StringBuffer sb1=new StringBuffer("AAA");
        StringBuffer sb2=new StringBuffer("BB");
        StringBuffer sb3=new StringBuffer("CCCC");
        StringBuffer sb4=new StringBuffer("D");
        StringBuffer sb5=new StringBuffer("EEEEE");
        MyComparator mc=new MyComparator();
        TreeSet ts=new TreeSet(mc);
        ts.add(sb1);
        ts.add(sb2);
        ts.add(sb3);
        ts.add(sb4);
        ts.add(sb5);
        System.out.println(ts);
    }
}
```

EX:

```
import java.util.*;
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Student s1=(Student)obj1;
        Student s2=(Student)obj2;

        int val=s1.saddr.compareTo(s2.saddr);
        return -val;
    }
}
class Student
{
    String sid;
    String sname;
    String saddr;

    Student(String sid, String sname, String saddr)
    {
      this.sid=sid;
      this.sname=sname;
      this.saddr=saddr;
    }
    public String toString()
    {
      return "["+sid+","+sname+","+saddr+"]";
    }
}
class Test
{
    public static void main(String[] args)
    {
      Student std1=new Student("S-111", "Durga", "Hyd");
      Student std2=new Student("S-222", "Anil", "Chennai");
      Student std3=new Student("S-333", "Rahul", "Banglore");
      Student std4=new Student("S-444", "Rameshh", "Pune");
      MyComparator mc=new MyComparator();
      TreeSet ts=new TreeSet(mc);
      ts.add(std1);
      ts.add(std2);
      ts.add(std3);
      ts.add(std4);
      System.out.println(ts);
    }
}
```

Q) <u>In Java applications, if we provide both implicit Sorting through Comparable and explicit sorting through Comparator to the TreeSet object at a time then which Sorting logic would be preferred by TreeSet inorder to Sort elements?</u>

If we provide both implicit Sorting through Comparable and Explicit Sorting through Comparator to the TreeSet object at a time then TreeSet will take explicit Sorting through Comparator to sort all the elements.

<u>EX:</u>

```
import java.util.*;
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        Customer cust1=(Customer)obj1;
        Customer cust2=(Customer)obj2;

        int val=cust1.caddr.compareTo(cust2.caddr);
        return -val;
    }
}
class Customer implements Comparable
{
    String cid;
    String cname;
    String caddr;

    Customer(String cid, String cname, String caddr)
    {
        this.cid=cid;
        this.cname=cname;
        this.caddr=caddr;
    }
    public String toString()
    {
        return "["+cid+","+cname+","+caddr+"]";
    }
    public int compareTo(Object obj)
    {
        Customer cust=(Customer)obj;
        int val=this.caddr.compareTo(cust.caddr);
        return val;
    }
}
class Test
{
    public static void main(String[] args)
    {
        Customer c1=new Customer("C-111", "Durga", "Hyd");
        Customer c2=new Customer("C-222", "Anil", "Chennai");
        Customer c3=new Customer("C-333", "Rahul", "Banglore");
        Customer c4=new Customer("C-444", "Ramesh", "Pune");
        MyComparator mc=new MyComparator();
        TreeSet ts=new TreeSet(mc);
        ts.add(c1);
        ts.add(c2);
        ts.add(c3);
        ts.add(c4);
        System.out.println(ts);
    }
}
```

Queue:

- It was introduced in JDK 5.0 version.
- It is a direct child interface to Collection Interface.
- It able to arrange all the elements as per FIFO [First In First Out], but, it is possible to change this algorithm as per our requirement.
- It able to allow duplicate elements.
- It is not following Insertion order.
- It is following Sorting order.
- It will not allow null values, if we add null value then JVM will rise an Exception like java.lang.NullPointerException.
- It will not allow heterogeneous elements, if we add heterogeneous elements then JVM will rise an exception like java.lang.ClassCastException.
- It able to allow only homogeneous elements.
- It able to allow comparable objects bydefault, if we add non comparable objects then JVM will rise an exception like java.lang.ClassCastException.
- If we want to add non comparable objects then we have to use Comparator.
- It able to manage all elements prior to process.

Methods:

- **public void offer(Object obj)**
  It can be used to insert the specified element to Queue.

- **public Object peek()**
  It can be used to return head element of the Queue.

- **public Object element()**
  It can be used to return head element of the Queue

Note: If we access peek() method on an empty Queue then peek() will return "null" value. If we access element() method on an empty Queue then element() method will rise an exception like java.util.NoSuchElementException

- **public Object poll()**
  It can be used to return and remove head element from Queue.

- **public Object remove()**

It can be used to return and remove head element from Queue.

Note: If we access poll() method on an empty Queue then poll() method will return "null" value. If we access remove() method on an empty Queue then remove() method will rise an exception like "java.util.NoSuchElementException".

EX:

```java
import java.util.*;
class Test
{
   public static void main(String[] args)
   {
      PriorityQueue q=new PriorityQueue();
      q.offer("A");
      q.offer("B");
      q.offer("C");
      q.offer("D");
      q.offer("E");
      q.offer("F");
      System.out.println(q);
      System.out.println(q.peek());
      System.out.println(q);
      System.out.println(q.element());
      System.out.println(q);
      /*
      PriorityQueue q1=new PriorityQueue();
      System.out.println(q1.peek());--> Null
      System.out.println(q1.element());--> Exception
      */
      System.out.println(q.poll());
      System.out.println(q);
      System.out.println(q.remove());
      System.out.println(q);
      /*
      PriorityQueue q1=new PriorityQueue();
      System.out.println(q1.poll());--> Null
      System.out.println(q1.remove());-->Exception
      */
   }
}
```

PriorityQueue:

- It was introduced in JDK 5.0 version.
- It is not Legacy Collection.
- It is a direct implementation class to Queue interface.

- It able to arrange all the elements prior to processing on the basis of the priorities.
- It able to allow duplicate elements.
- It is not following Insertion order.
- It is following Sorting order.
- It will not allow null values, if we add null value then JVM will rise an Exception like java.lang.NullPointerException.
- It will not allow heterogeneous elements, if we add heterogeneous elements then JVM will rise an exception like java.lang.ClassCastException.
- It able to allow only homogeneous elements.
- It able to allow comparable objects by default, if we add non comparable objects then JVM will rise an exception like java.lang.ClassCastException.
- If we want to add non comparable objects then we have to use Comparator.
- Its initial capacity 11 elements.
- It is not synchronized .
- No method is synchronized in PriorityQueue.
- It allows more than one thread at a time to access data.
- It follows parallel execution.
- It able to reduce application execution time.
- It able to improve application performance.
- It is not giving guarantee for Data consistency.
- It is not threadsafe.

Constructors:

- **public PriorityQueue()**

  It able to create an empty PriorityQueue object

  EX: PriorityQueue p = new PriorityQueue();

- **public PriorityQueue(int capacity)**

  It can be used to create an empty Queue with the specified capacity.

  EX: PriorityQueue p = new PriorityQueue(20);

- **public PriorityQueue(int capacity,Comparator c)**

  It able to create an empty PriorityQueue with explicit sorting logic throug COmparator and the specified capacity.

  EX:
  ```
  MyComparator mc = new MyComparator();
  PriorityQueue p = new PriorityQueue(20,mc);
  ```

- **public PriorityQueue(SortedSet ss)**

  It able to create PriorityQueue object with all the elements of the specified SortedSet.

EX:

```
TreeSet ts=new TreeSet();
ts.add("B");
ts.add("E");
ts.add("C");
ts.add("A");
ts.add("D");
System.out.println(ts);
PriorityQueue p = new PriorityQueue(ts);
System.out.println(p);
```

- **public PriorityQueue(Collection c)**

  It able to create PriorityQueue object with all the elements of the specified Collection.

  EX:

```
ArrayList al = new ArrayList();
al.add("A");
al.add("B");
al.add("C");
al.add("D");
System.out.println(al);
PriorityQueue p = new PriorityQueue(al);
System.out.println(p);
```

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    PriorityQueue p=new PriorityQueue();
    p.add("A");
    p.add("D");
    p.add("B");
    p.add("C");
    p.add("F");
    p.add("E");
    System.out.println(p);
    p.add("B");
    System.out.println(p);
    //p.add(null);-->NullPointerException
    //p.add(new Integer(10));->ClassCastException
    //p.add(new StringBuffer("BBB"));->ClassCastException
  }
}
```

Map:

- It was introduced in JDK 1.2 version.
- It is not child interface to Collection Interface.
- It able to arrange all the elements in the form of Key-value pairs.
- In Map, both keys and values are objects.
- Duplicates are not allowed at keys, but values may be duplicated.
- Only one null value is allowed at keys side, but, any number of null values are allowed at values side.
- Both keys and Values are able to allow heterogeneous elements.
- Insertion order is not followed.
- Sorting order is not followed.

Methods:

- **public void put(Object key, Object value)**
  It will add the specified key-value pair to Map.
- **public void putAll(Map m)**
  It will add all key-value pairs of the specified map to the present Map object.
- **public Object get(Object key)**
  It will return value of the specified key.
- **public Object remove(Object key)**
  It will remove a key-value pair from Map on the basis of the specified key.
- **public int size()**
  It will return number of key-value pairs of a Map
- **public boolean containsKey(Object key)**
  It will check whether the specified key is existed or not at keys side.
- **public boolean cotainsValue(Object key)**
  It will check whether the specified value is available or not at values side.
- **public Set keySet()**
  It will return all keys in the form of a Set.

- **public Collection values()**

It will return all values in the form of a Collection object.

- **public boolean isEmpty()**

It will check whether the Map object is empty or not, if the present map object is empty then it will return true value otherwise it will return false value.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    HashMap hm=new HashMap();
    hm.put("A","AAA");
    hm.put("B","BBB");
    hm.put("C","CCC");
    hm.put("D","DDD");
    hm.put("E","EEE");
    System.out.println(hm);
    HashMap hm1=new HashMap();
    hm1.put("X","XXX");
    hm1.put("Y","YYY");
    System.out.println(hm1);
    hm.putAll(hm1);
    System.out.println(hm);
    System.out.println(hm.get("B"));
    System.out.println(hm.remove("E"));
    System.out.println(hm.size());
    System.out.println(hm.isEmpty());
    System.out.println(hm.containsKey("D"));
    System.out.println(hm.containsValue("DDD"));
    System.out.println(hm.keySet());
    System.out.println(hm.values());
  }
}
```

HashMap:

- **It was introduced in JDK1.2 version.**
- **It is not Legacy**

- It is an implementation class to Map interface.
- It able to arrange all the elements in the form of Key-value pairs.
- In HashMap, both keys and values are objects.
- Duplicates are not allowed at keys, but values may be duplicated.
- Only one null value is allowed at keys side, but, any number of null values are allowed at values side.
- Both keys and Values are able to allow heterogeneous elements.
- Insertion order is not followed.
- Sorting order is not followed.
- Its internal data structure is "Hashtable".
- Its initial capacity is 16 elements.
- It is not synchronized
- No method is synchronized in HashMap
- It allows more than one thread to access data.
- It follows parallel execution.
- It will reduce application execution time.
- It will improve application performance.
- It is not giving guarantee for data consistency.
- It is not threadsafe.

Constructors:

- public HashMap()
- public HashMap(int capacity)
- public HashMap(int capacity, float fill_Ratio)
- public HashMap(Map m)

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    HashMap hm=new HashMap();
    hm.put("A","AAA");
    hm.put("B","BBB");
    hm.put("C","CCC");
    hm.put("D","DDD");
    hm.put("E","EEE");
    System.out.println(hm);
    hm.put("B","FFF");
    System.out.println(hm);
    hm.put("F","CCC");
    System.out.println(hm);
    hm.put(null,"GGG");
```

```
        hm.put(null,"HHH");
        hm.put("G",null);
        hm.put("H",null);
        System.out.println(hm);
        hm.put(new Integer(10), new Integer(20));
        System.out.println(hm);
    }
}
```

LinkedHashMap:

Q) What are the differences between *HashMap* and *LinkedHashMap?*

- **HashMap was introduced in JDK 1.2 version.**
  LinkedHashMap was introduced in JDK 1.4 version.

- **HashMap is not following insertion order.**
  LinkedHashMap is following insertion order.

- **HashMap internal data structure is Hashtable.**
  LinkedHashMap internal data structure is Hashtable+LinkedList

EX:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        HashMap hm=new HashMap();
        hm.put("A","AAA");
        hm.put("B","BBB");
        hm.put("C","CCC");
        hm.put("D","DDD");
        hm.put("E","EEE");
        System.out.println(hm);

        LinkedHashMap lhm=new LinkedHashMap();
        lhm.put("A","AAA");
        lhm.put("B","BBB");
        lhm.put("C","CCC");
        lhm.put("D","DDD");
        lhm.put("E","EEE");
        System.out.println(lhm);
```

```
    }
}
```

IdentityHashMap:

Q) What are the differences between *HashMap* and *IdentityHashMap?*

- **HashMap was introduced in JDK 1.2 version.**
  IdentityHashMap was introduced in JDK 1.4 version.

- **HashMap and IdentityhashMap are not allowing duplicate keys, to check duplicate keys HashMap will use equals(-) method, but, IdentityHashMap will use '==' operator.**

Note: '==' operator will perform references comparison always, but, equals() method was defined in Object class initially, later on it was overridden in String class and in all wrapper classes in order to perform contents comparison.

EX:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        Integer in1=new Integer(10);
        Integer in2=new Integer(10);
        HashMap hm=new HashMap();
        hm.put(in1,"AAA");
        hm.put(in2,"BBB");// in2.equals(in1)-->true
        System.out.println(hm);// {10=BBB}

        IdentityHashMap ihm=new IdentityHashMap();
        ihm.put(in1, "AAA");
        ihm.put(in2, "BBB");// in2 == in1 --> false
        System.out.println(ihm);// {10=AAA, 10=BBB}
    }
}
```

WeakHashMap:

Q) What is the difference between *HashMap* and

WeakHashMap?

- Once if we add an element to HashMap then HashMap is not allowing Garbage Collector to destroy its objects.
- Even if we add an element to WeakHashMap then WeakHashMap is able to allow Garbage Collector to destroy elements.

EX:

```java
import java.util.*;
class A
{
  public String toString()
  {
    return "A";
  }
}
class Test
{
  public static void main(String[] args)
  {
    A a1=new A();
    HashMap hm=new HashMap();
    hm.put(a1, "AAA");
    System.out.println("HM Before GC  :"+hm);// {A=AAA}
    a1=null;
    System.gc();
    System.out.println("HM After GC   :"+hm);// {A=AA}

    A a2=new A();
    WeakHashMap whm=new WeakHashMap();
    whm.put(a2, "AAA");
    System.out.println("WHM Before GC :"+whm);// {A=AAA}
    a2=null;
    System.gc();
    System.out.println("WHM After GC  :"+whm);// {}

  }
}
```

NOTE: In Java applications, Garbage Collector will destroy objects internally. In java applications, it is possible to destroy objects explicitly by activating GarbageCollector, for this, we have to use the following two steps.

- Nullify the respective object reference.
- Access System.gc() method, it will access finalize() method internally just before destroying object.

EX:

```
class A {
    A() {
            System.out.println("Object Creating");
    }
    public void finalize() {
        System.out.println("Object Destroying");
    }
}
class Test {
    public static void main(String[] args) {
        A a = new A();
        a = null;
        System.gc();
    }
}
```

SortedMap:

- It was introduced in JDK1.2 version.
- It is a direct child interface to Map interface
- It able to allow elements in the form of Key-Value pairs, where both keys and values are objects.
- It will not allow duplicate elements at keys side, but, it able to allow duplicate elements at values side.
- It will not follow insertion order.
- It will follow sorting order.
- It will not allow null values at keys side. If we are trying to add null values at keys side then JVM will rise an exception like java.lang.NullPointerException.
- It will not allow heterogeneous elements at keys side, if we are trying add heterogeneous elements then JVM will rise an exception like java.lang.ClassCastException.
- It able to allow only comparable objects at keys side by default, if we are trying to add non comparable objects then JVM will rise an exception like java.lang.ClassCastException.
- If we want to add non comparable objects then we must use Comparator.

Methods:

public Object firstKey()
public Object lastKey()
public SortedMap headMap(Object key)
public SportedMap tailMap(Object key)
public SortedMap subMap(Object obj1, Object obj2)

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    TreeMap tm=new TreeMap();
    tm.put("B", "BBB");
    tm.put("E", "EEE");
    tm.put("D", "DDD");
    tm.put("A", "AAA");
    tm.put("F", "FFF");
    tm.put("C", "CCC");
    System.out.println(tm);
    System.out.println(tm.firstKey());
    System.out.println(tm.lastKey());
    System.out.println(tm.headMap("D"));
    System.out.println(tm.tailMap("D"));
   System.out.println(tm.subMap("B", "E"));
  }
}
```

NavigableMap:

It was introduced in JAVA6 version, it is a child interface to SortedMap and it has defined methods to provide navigations over the elements.

Methods:

- **public Object CeilingKey(Object obj)**
- **public Object higherKey(Object obj)**
- **public Object floorKey(Object obj)**
- **public Object lowerKey(Object obj)**
- **public NavigableMap descendingMap()**
- **public Map.Entry pollFirstEntry()**
- **public Map.Entry pollLastEntry()**

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
```

```
        TreeMap tm=new TreeMap();
        tm.put("A", "AAA");
        tm.put("B", "BBB");
        tm.put("C", "CCC");
        tm.put("D", "DDD");
        tm.put("E", "EEE");
        tm.put("F", "FFF");
        System.out.println(tm);
        System.out.println(tm.descendingMap());
        System.out.println(tm.ceilingKey("D"));
        System.out.println(tm.higherKey("D"));
        System.out.println(tm.floorKey("D"));
        System.out.println(tm.lowerKey("D"));
        System.out.println(tm.pollFirstEntry());
        System.out.println(tm.pollLastEntry());
        System.out.println(tm);
    }
}
```

TreeMap:

- It was introduced in JDK 1.2 version.
- It is not Legacy.
- It is an implementation class to Map, SoortedMap and NavigableMap interfaces.
- It able to allow elements in the form of Key-Value pairs, where both keys and values are objects.
- It will not allow duplicate elements at keys side, But, it able to allow duplicate elements at values side.
- It will not follow insertion order.
- It will follow sorting order.
- It will not allow null values at keys side. If we are trying to add null values at keys side then JVM will rise an exception like java.lang.NullPointerException.
- It will not allow heterogeneous elements at keys side, if we are trying add heterogeneous elements then JVM will rise an exception like java.lang.ClassCastException.
- It able to allow only comparable objects at keys side by default, if we are trying to add non comparable objects then JVM will rise an exception like java.lang.ClassCastException.
- If we want to add non comparable objects then we must use Comparator.
- Its internal data Structer is "Red-Black Tree".
- It is not synchronized.
- No methods are synchronized in TreeMap.
- It allows more than one thread to access data.
- It will follow parallel execution.
- It will reduce execution time.
- It will improve application performance.

- It is not giving guarantee for Data Consistency.
- It is not threadsafe.

Constructors:

- public TreeMap()
- public TreeMap(Comparator c)
- public TreeMap(SortedMap sm)
- public TreeMap(Map m)

EX:

```java
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    TreeMap tm=new TreeMap();
    tm.put("A", "AAA");
    tm.put("B", "BBB");
    tm.put("C", "CCC");
    tm.put("D", "DDD");
    System.out.println(tm);
    tm.put("B", "EEE");
    System.out.println(tm);
    tm.put("E", "CCC");
    System.out.println(tm);
    //tm.put(null, "EEE");-->NullPointerException
    tm.put("F",null);
    System.out.println(tm);
    //tm.put(new Integer(10), new Integer(20));-->CCE
    System.out.println(tm);
    tm.put("G", new Integer(20));
    System.out.println(tm);
    //tm.put(new StringBuffer("BBB"), "GGG");-->CCE
    tm.put("H", new StringBuffer("HHH"));
    System.out.println(tm);
  }
}
```

Hashtable:

Q) What are the differences between *HashMap* and *Hashtable?*

- HashMap was introduced in JDK 1.2 version.
  Hashtable was introduced in JDK 1.0 version.

- HashMap is not Legacy.
  Hashtable is Legacy.

- In HashMap, one null value is allowed at keys side and any number of null values are allowed at values side.
  In case of Hashtable, null values are not allowed at both keys and values side.

- HashMap is not synchronized.
  Hashtable is synchronized.

- No method is synchronized in HashMap.
  Almost all the methods are synchronized in Hashtable

- HashMap allows more than one thread to access data.
  Hashtable allows only one thread at a time to access data.

- HashMap follows parallel execution.
  Hashtable follows sequential execution.

- HashMap will reduce execution time.
  Hashtable will increase execution time.

- HashMap will improve application performance.
  Hashtable will reduce application performance.

- HashMap will not give guarantee for data consistency.
  Hashtable will give guarantee for data consistency

- HashMap is not threadsafe.
  Hashtable is threadsafe.

EX:

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    HashMap hm=new HashMap();
```

```
        hm.put("A", "AAA");
        hm.put("B", "BBB");
        hm.put("C", "CCC");
        hm.put("D", "DDD");
        System.out.println(hm);
        hm.put(null, "EEE");
        hm.put(null, "FFF");
        hm.put("E",null);
        hm.put("F", null);
        System.out.println(hm);
        System.out.println();
        Hashtable ht=new Hashtable();
        ht.put("A", "AAA");
        ht.put("B", "BBB");
        ht.put("C", "CCC");
        ht.put("D", "DDD");
        System.out.println(ht);
        //ht.put(null, "EEE");-->NullPointerException
        //ht.put("E", null);-->NullPointerException
    }
}
```

Properties:

In Java applications, if we have any data which we want to change frequently then we have to manage that type of data in a properties file, otherwise we have to perform recompilation on every modification.

The main purpose of properties files in java applications is,

- **To manage labels of the GUI components in GUI appl.**
- **To manage locale respective messages in I18N Appl.**
- **To manage exception messages in Exception handling.**
- **To manage validation messages in Data validations.**

EX:

user.properties

```
 uname=User Name
upwd=User password
uname.required=User Name is Required
upwd.required=User Password is required
```

exception.insufficientfunds=Funds are not sufficient in your Account.

In Java applications, to represent data of a particular properties file we have to use java.util.Properties class.

To get data from a particular properties file to Properties object we have to use the following steps.

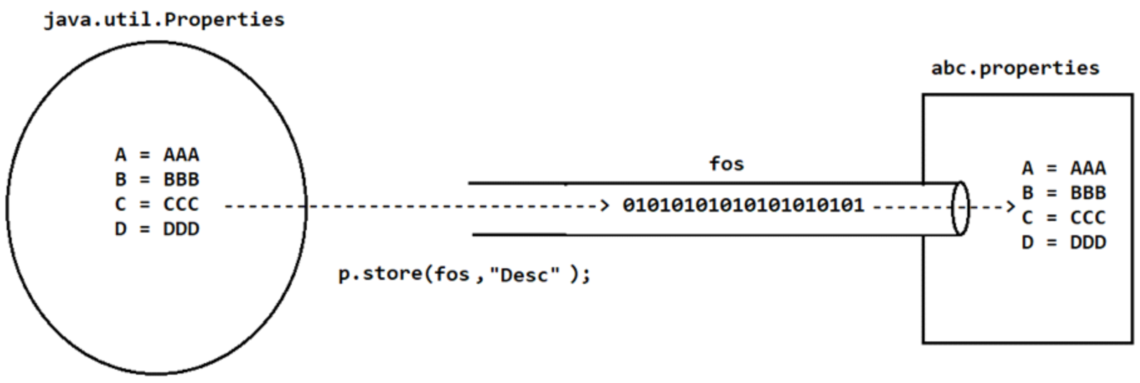
- Create Properties file with the data in the form of Key-value pairs.
- Create Properties class object.
- Create FileInputStream to get data from properties file.
- Load data from FileInputStream to Properties object by using the following method.
  public void load(FileInputStream fis)
- Get data from Properties object by using the following method.
  public String getProperty(String key)

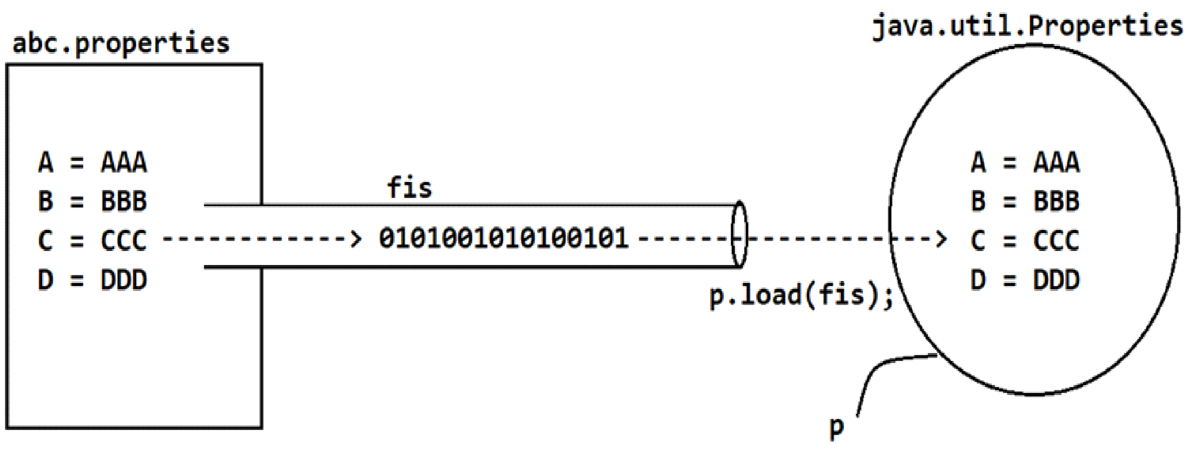
To send data from Properties object to properties file we have to use the following steps.

- Create Properties class object.
- Set data to Properties object by using the following method.
  public void setProperty(String key, String val)
- Create FileOutputStream with the target file.
- Store Properties object data to FileOutputStream by using the following method.
  public void store(FileOutputStream fos, String des)

db.properties

```
driver_Class=sun.jdbc.odbc.JdbcOdbcDriver
driver_URL=jdbc:odbc:dsn_name
db_User=system
db_Password=durga
```





Test.java

```
import java.util.*;
import java.io.*;
class Test
{
  public static void main(String[] args)throws Exception
  {
    Properties p=new Properties();
    FileInputStream fis=new FileInputStream("db.properties");
    p.load(fis);
    System.out.println("JDBC Parameters");
    System.out.println("--------------------");
    System.out.println("Driver_Class  :"+p.getProperty("driver_Class"));
    System.out.println("Driver_URL    :"+p.getProperty("driver_URL"));
    System.out.println("DB_User       :"+p.getProperty("db_User"));
    System.out.println("DB_PAssword   :"+p.getProperty("db_Password"));
  }
}
```

Test1.java

```
import java.util.*;
import java.io.*;
class Test1
{
  public static void main(String[] args)throws Exception
  {
    Properties p=new Properties();
    p.setProperty("uname","Durga");
    p.setProperty("upwd", "durga123");
    p.setProperty("uqual", "M Tech");
    p.setProperty("uemail", "durga@durgasoft.com");
```

```
        p.setProperty("umobile","91-9988776655");

        FileOutputStream fos=new FileOutputStream("user.properties");
        p.store(fos,"User Details");
    }
}
```

Generics

What is the Requirement to use Generics in Java Applications?

- To represent one thing if we allow only one data type then it is type safe operation.

  EX: Arrays are providing type safe operation.

  ```
  Student[] std = new Student[3];
  std[0] = new Student();----> Valid
  std[1] = new Student();----> Valid
  std[2] = new Customer();---> Invalid
  ```

To represent one thing if we allow more than one type then it is Type unsafe operation.

EX: Collection objects are providing type Unsafe operation, because, Collection objects are able to allow heterogeneous elements.

```
 ArrayList al = new ArrayList();
 al.add("AAA");
 al.add(new Employee());
 al.add(new Integer(10));
 al.add(new StringBuffer("CCC"));
```

In Collections to improve typedness or to provide type safe operations then we have to use "Generics".

- If we add elements to the Collection object and if we want to retrieve elements from Collection objects we must perform type casting, because, get(--) methods of Collections are able to retrieve elements in the form of java.lang.Object type.

EX:

```
ArrayList al = new ArrayList();
al.add("A");
al.add("B");
al.add("C");
al.add("D");
System.out.println(al);
String str = al.get(2);---> Compilation Error, Incompatible types.
String str = (String)al.get(2);--> Valid
```

If we use Generics along with Collections then it is not required to type cast while iterating elements.

The main intention of Generics is,

- To provide typesafe operations in Collections.
- To avoid Type casting while retrieving elements from Collections.

Generics is a Type parameter specified along with Collections in order to fix a particular type of elements to add.

Syntax:

Collection_Name<Generic_Type> ref = new Collection_Name<GenericType>([Params]);

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

The above ArrayList object is able to add only String type elements, if we are trying to add any other elements then Compiler will rise an error.

EX:

```
import java.util.*;
class Test
{
   public static void main(String[] args)
   {
      ArrayList<String> al=new ArrayList<String>();
      al.add("A");
      al.add("B");
      al.add("C");
      al.add("D");
      al.add("E");
      System.out.println(al);
      al.add(new Integer(10));---> Error
      System.out.println(al);
   }
}
```

Before JDK 5.0 version, ArrayList class is

```
public class ArrayList
{
   public void add(Object obj)
   {
   }
   public Object get(int index)
   {
   }
    -----
    -----
}
```

From JDK 5.0 version ArrayList class is

```
public class ArrayList<T>
{
```

```
    public void add(T t)
    {
    }
    public T get(int index)
    {
   }
}
```

In the above ArrayList we can provide any user defined data type to T.

EX 1: ArrayList<String> al = new ArrayList<String>();
Here ArrayList class is converted internally as

```
public class ArrayList<String>
{
    public void add(String t)
    {
    }
    public String get(int index)
    {
   }
}
```

Here add(-) method is able to take only String elements to add to the ArrayList and get() method will return only String elements.

al.add("A");---> Valid
al.add("B");---> Valid
al.add("C");---> Valid
al.add(new Integer(10));---> Invalid.
String str = al.get(2);

EX 2: ArrayList<Integer> al = new ArrayList<Integer>();
Here ArrayList class is converted internally like below.

```
public class ArrayList<Integer>
{
    public void add(Integer t)
    {
    }
    public Integer get(int index)
    {
   }
}
```

Here add(-) method is able to take only Integer elements to add to the ArrayList and get() method will return only Integer elements.

```
al.add(new Integer(10));---> Valid
al.add(new Integer(20));---> Valid
al.add(30);---> Valid, Autoboxing
al.add("AAA");---> Invalid.
Integer in=al.get(1);--> valid
int i=al.get(2);---> Valid, Auto-Unboxing
```

If any java class is having Generic type parameter at its declaration then that class is called as "Generic Class".

NOTE: "Generics" feature is provided by JAVA along with its JDK5.0 version.

EX:
```
class Account<T>
{
}

Account<Savings> acc=new Account<Savings>();
Account<Current> acc=new Account<Current>();
```

In the above Generic class declaration, we can use any valid java identifier as Type parameter.

EX:
```
class Account<X>
{
}
```

Status: Valid

EX:
```
class Account<Durga>
{
}
```

Status: Valid

In the above Generic class declaration, we can use any no of Type parameters by provide ',' seperator.

EX:
```
class HashMap<K,V>
```

```
{
}
```

WHere 'K' is key.
Where 'V' is value.

EX:

```
class Account<T>
{
    T obj;
    public void set(T obj)
    {
        this.obj=obj;
    }
    public T get()
    {
        return obj;
    }
    public void display_Type()
    {
        System.out.println(obj.getClass().getName());
    }
}
class Test
{
    public static void main(String[] args)
    {
        Account<String> acc=new Account<String>();
        acc.set("Savings_Account");
        System.out.println(acc.get());
        acc.display_Type();
    }
}
```

Bounded Types:

In Generic Classes, we can bound the type parameter for a particular range by using "extends" keyword.

EX 1:

```
class Test<T>
{
}
```

It is unbounded type, we can pass any type of parameter.

```
Test<String> t=new Test<String>();
Test<Integer> t=new Test<Integer>();
```

EX 2:

```
class Test<T extends X>
{
}
```

If X is a class type then we can pass either X type elements or sub class elements of X type as parameter types.

EX:

```
class Payment
{
}
class CashPayment extends Payment
{
}
class CardPayment extends Payment
{
}
class Bill<T extends Payment>
{
}

Bill<Payment> bill = new Bill<Payment>(); ---> Valid
Bill<CashPayment> bill = new Bill<CashPayment>();--> Valid
Bill<CardPayment> bill = new Bill<CardPayment>();---> Valid
```

EX:

```
class Test<T extends Number>
{
}
Test<Number> t=new Test<Number>()---> Valid
Test<Integer> t=new Test<Integer>();----> Valid
Test<Float> t=new Test<Float>();----> Valid
Test<String> t=new Test<String>();---> Invalid
```

If 'X' is an interface then we are ABle to pass either X type elements of its implementation class types as parameter.

<u>EX:</u>

```
interface Java
{
}
class CoreJava implements Java
{
}
class AdvJava implements Java
{
}
class Course<T extends Java>
{
}

Course<Java> crs1 = new Course<Java>(); --->Valid
Course<CoreJava> crs2 = new Course<CoreJava>();--> Valid
Course<AdvJava> crs3 = new Course<AdvJava>();---> Valid
```

<u>NOTE:</u> Bouded parameters are not allowing "implements" keyword and "super" keyword.

```
class Test<T implements Runnable>
{
}
```

Status: Invalid

<u>EX:</u>

```
class Test<T super String>
{
}
```

Status: Invalid

In Generic classes, we can use more than one type as bounded parameter by using '&' symbol.

<u>EX:</u>

```
class Test<T extends Number & Serializable>
{
}
```

Here Test class is able to allow the elements which must be either same as NUmber or sub classes to the number and which must be the implementations of Serializable interface.

Test<Integer> t=new Test<Integer>();--> Valid
Test<Float> t=new Test<Float>();--> Valid
Test<String> t=new Test<String>--> Invalid.

EX:
```
clas Test<T extends Number & Thread>
{
}
```

Status: Invalid

Reason: extends keyword will not allow two class types at a time.

EX:
```
class Test<T extends Runnbale & Serializable>
{
}
```

Statis: valid.

Reason: extends keyword is able to allow more than one interface type.

EX:
```
class Test<T extends Number & Runnable>
{
}
```
Status: Valid.

Reson: Extends is able to allow both class type and interface type , but, first we have to specify class type then we ahve to specify Interface type.

EX:
```
class Test<T extends Runnable & Number>
{
}
```
Status: Invalid.
Reason: extends keyword allows first class_Name then interface_Name
EX:
```
import java.util.*;
class Account
{
    String accNo;
    String accName;
    String accType;

    Account(String accNo, String accName, String accType)
    {
        this.accNo=accNo;
        this.accName=accName;
        this.accType=accType;
    }

}
class Bank
{
    public ArrayList<Account> getAccountsList(ArrayList<Account> al)
    {
        al.add(new Account("a111", "AAA", "Savings"));
        al.add(new Account("b111", "BBB", "Savings"));
        al.add(new Account("c111", "CCC", "Savings"));
        al.add(new Account("d111", "DDD", "Savings"));
        return al;
    }
}
class Test {
    public static void main(String[] args)
    {
        ArrayList<Account> al=new ArrayList<Account>();
        Bank bank=new Bank();
        ArrayList<Account> acc_List=bank.getAccountsList(al);
        System.out.println("ACCNO\tACCNAME\tACCTYPE");
        System.out.println("------------------------------");
        for(Account acc: acc_List)
        {
            System.out.println(acc.accNo+"\t"+acc.accName+"\t"+acc.accType);
        }
    }
}
```