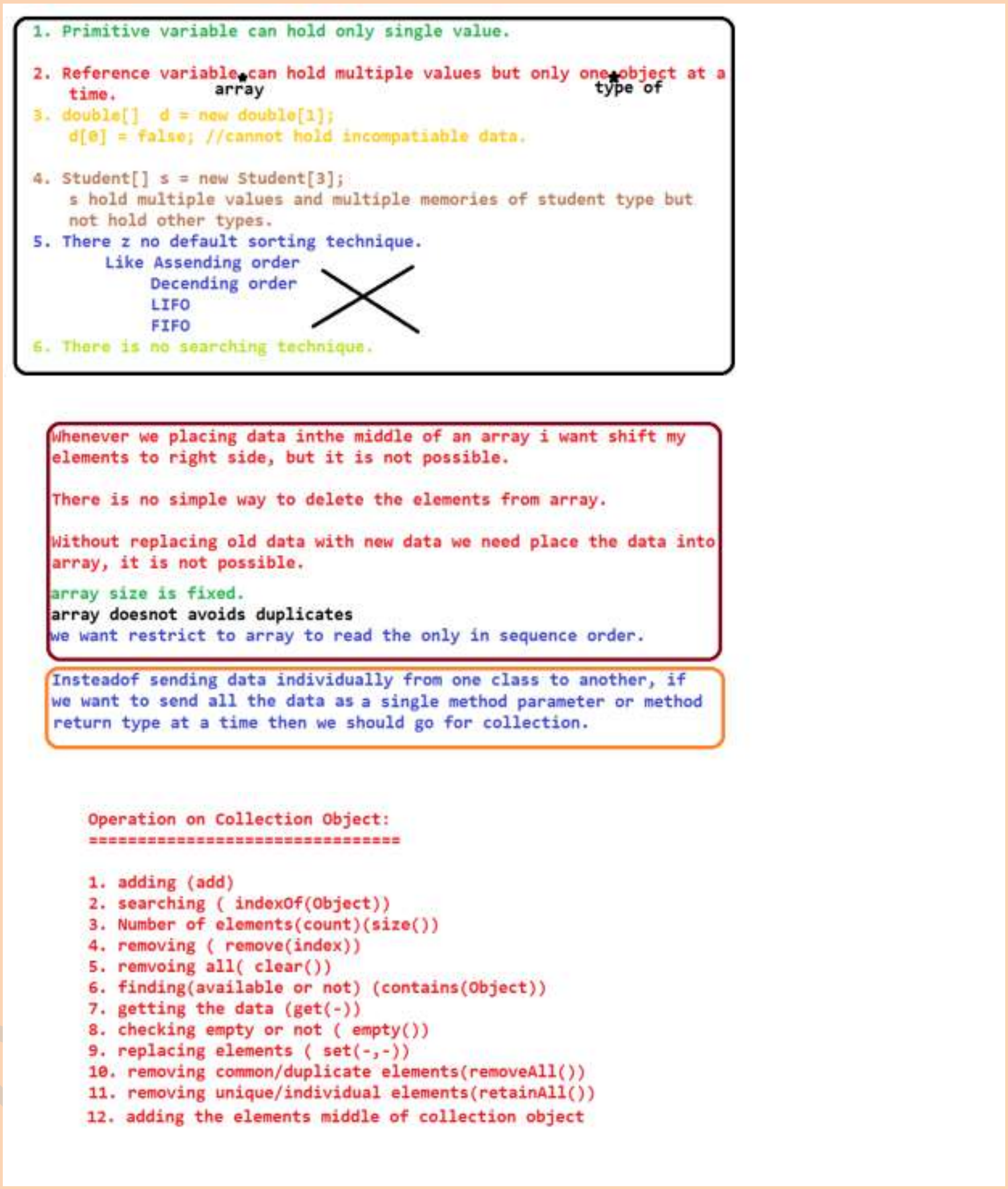
COLLECTION FRAME WORK

Drawbacks of Arrays:

- Arrays are fixed in size.
  - int a [] = new a [10]
  - Here “10” is size of an array.
- Once we creating an array there is no chance of increasing/decreasing its size based on our requirement.
- Array can hold only homogeneous data elements.
  - Teacher t[]=new Teacher[10];
- Here we can add only teacher object, but it cannot hold student, employee objects data.
- If we want overcome this problem we should go for Object array.
  - Object o[] =new Object[10];
  - o[1]=”rams”
  - Here “rams” is an string object
  - o[2]=new Teacher();
  - Here we did place the Teacher object.
- Arrays can allow the duplication code/data.
  - int a[]={10,20,6,34,10}
- Arrays don’t provide any low level services.
- Arrays don’t have any searching, sorting logic by default; we have to write the entire low level code/functionalities.
- To resolve these above problems we should go for collections framework.



1. Primitive variable can hold only single value.

2. Reference variable* can hold multiple values but only one* object at a time.
   array type of
3. double[] d = new double[1];
   d[0] = false; //cannot hold incompatiable data.

4. Student[] s = new Student[3];
   s hold multiple values and multiple memories of student type but not hold other types.
5. There z no default sorting technique.
   Like Assending order
   Decending order
   LIFO
   FIFO
6. There is no searching technique.

Whenever we placing data inthe middle of an array i want shift my elements to right side, but it is not possible.

There is no simple way to delete the elements from array.

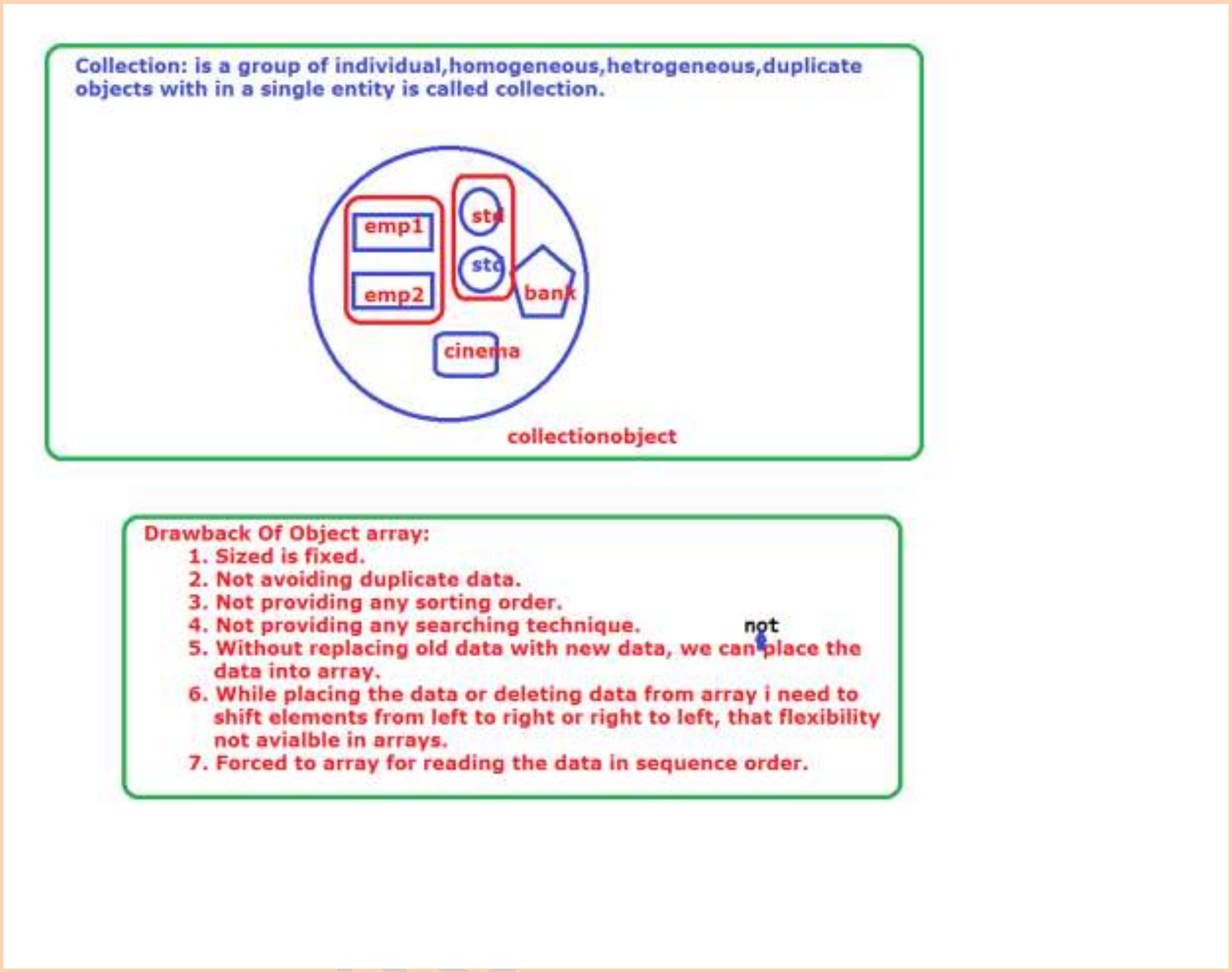
Without replacing old data with new data we need place the data into array, it is not possible.

array size is fixed.
array doesnot avoids duplicates
we want restrict to array to read the only in sequence order.

Insteadof sending data individually from one class to another, if we want to send all the data as a single method parameter or method return type at a time then we should go for collection.

Operation on Collection Object:
================================

1. adding (add)
2. searching (indexOf(Object))
3. Number of elements(count)(size())
4. removing (remove(index))
5. remvoing all(clear())
6. finding(available or not) (contains(Object))
7. getting the data (get(-))
8. checking empty or not (empty())
9. replacing elements (set(-,-))
10. removing common/duplicate elements(removeAll())
11. removing unique/individual elements(retainAll())
12. adding the elements middle of collection object


Collection: is a group of individual,homogeneous,hetrogeneous,duplicate objects with in a single entity is called collection.

Drawback Of Object array:

1. Sized is fixed.
2. Not avoiding duplicate data.
3. Not providing any sorting order.
4. Not providing any searching technique.
5. Without replacing old data with new data, we can not place the data into array.
6. While placing the data or deleting data from array i need to shift elements from left to right or right to left, that flexibility not avialble in arrays.
7. Forced to array for reading the data in sequence order.

Collection framework:

- ✓ Collection framework is a class library to handle group of objects.
- ✓ It is implemented in java.util package. It has been including in java 2.0.
- ✓ It is collection of classes and interface.
- ✓ Each and every class and interface having its own priority and advantages.

Advantages of Collection:

- The size must be increase dynamically, based on our application requirement.
    - We can overcome the problem of memory wastage.

  - We will get the application efficiency and performance.
  - The size will be not only increasing but also decreasing; we can see this nature in ArrayList in briefly.
- We collect/store different type and same type of objects in to one variable. That means we can store homogeneous and heterogeneous objects in our collection variable.
- Collection having readymade/predefine methods for manipulate the objects.
- Based on the requirement, we can store the duplication code and avoiding the duplication code or data redundancy.
- All Collection classes are implements Cloneable, Seriablizable interfaces.
- Both ArrayList and Vector classes implements RandomAccess interface also.
- All Collection class override the toString() of Object class.

Collection:

- **C**ollection is a root interface for representing a group of objects nothing but elements as a single entity.
- Collection doesn't allow duplicates and some are allow duplicate values.
- Collection are having order and some not having any order.
- There is no direct implementation of this interface.

Collections:

Collections is an utility class available in java.util package for defining several utility methods for collection objects.

Types of Collection Framework:

Based on the way of strong the objects, the collection framework is categorized into two approaches.

(a) Collection hierarchy.
(b) Map hierarchy.

Collection hierarchy:

Mainly it has been classified into three categories.

(a) List
(b) Set
(c) Queues

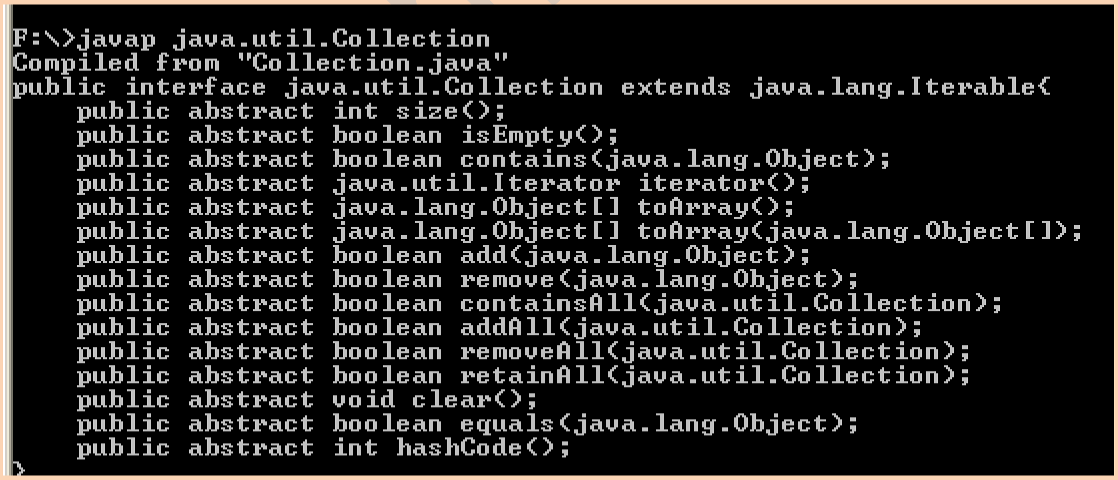
Map hierarchy:

In the map hierarchy in the objects will stored in the order of key-value pairs.

Collection interface:

- Collection is a root interface.
- It represents group of objects into single entity.
- It is having common methods, which can be applied on any objects.

Methods in collection interface:

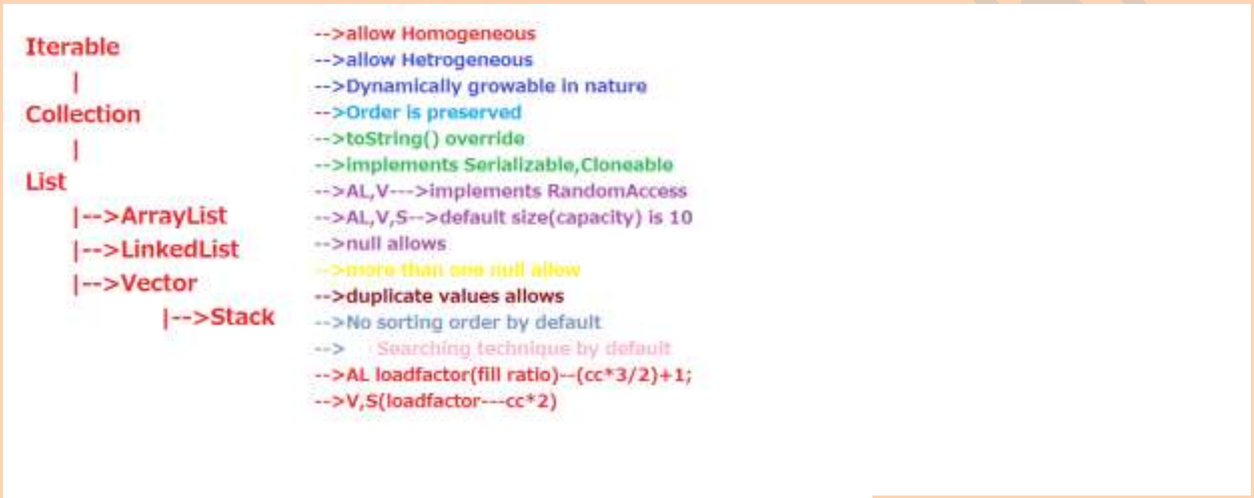


```
F:\>javap java.util.Collection
Compiled from "Collection.java"
public interface java.util.Collection extends java.lang.Iterable{
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean contains(java.lang.Object);
    public abstract java.util.Iterator iterator();
    public abstract java.lang.Object[] toArray();
    public abstract java.lang.Object[] toArray(java.lang.Object[]);
    public abstract boolean add(java.lang.Object);
    public abstract boolean remove(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean addAll(java.util.Collection);
    public abstract boolean removeAll(java.util.Collection);
    public abstract boolean retainAll(java.util.Collection);
    public abstract void clear();
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
}
```

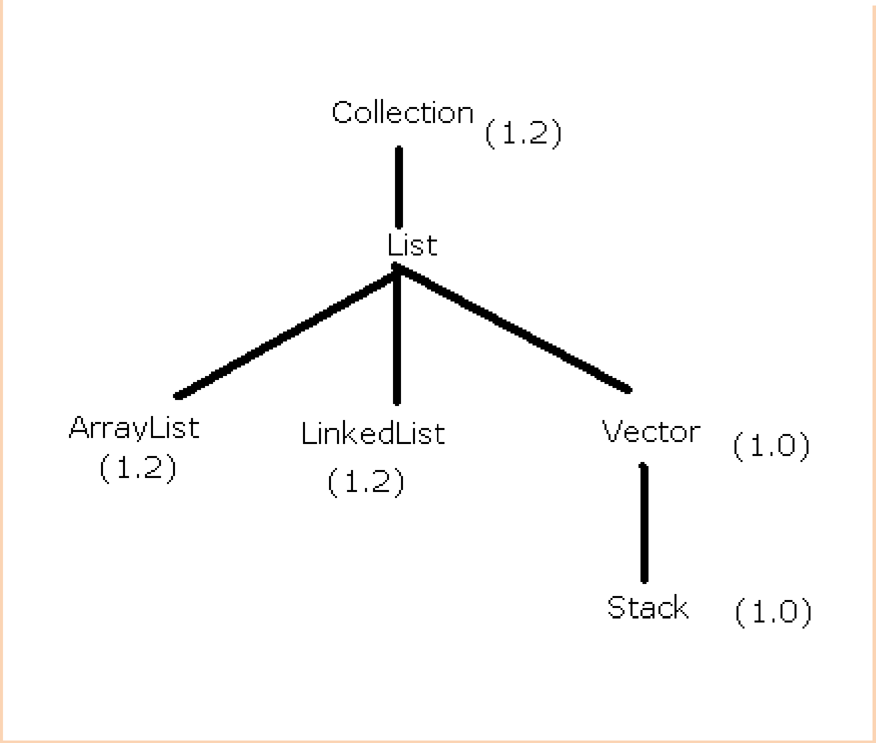
List interface:

- List is a child interface of Collection.

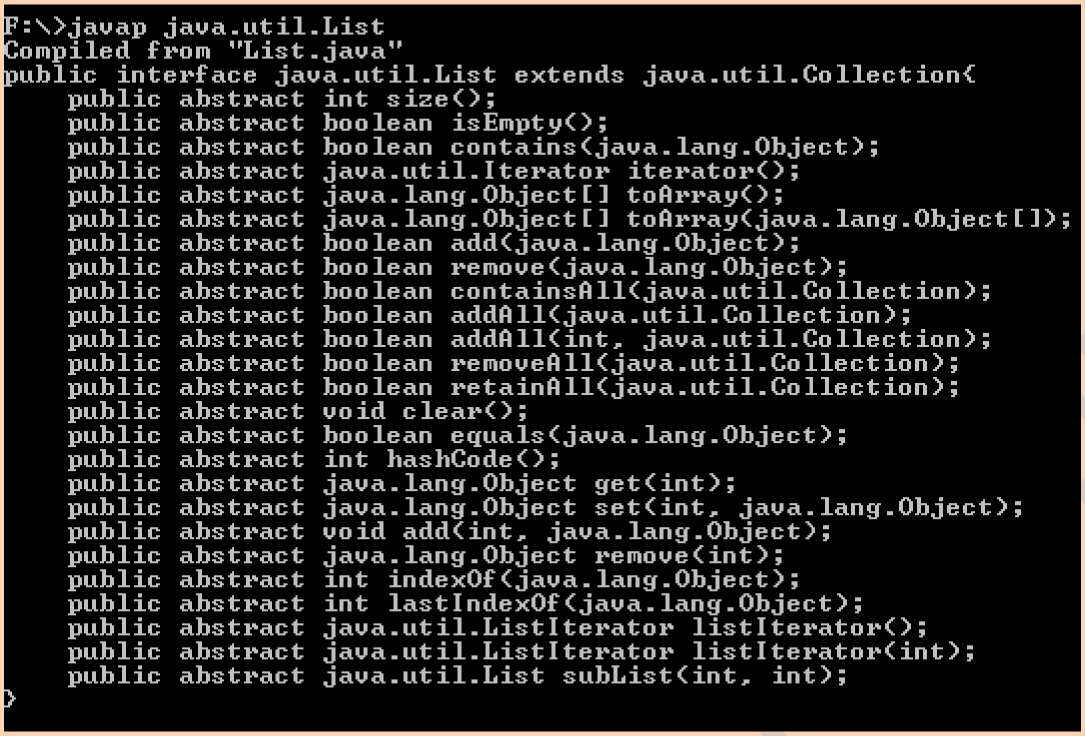
- It is used for group of individual objects as single entity.
- It can allow the duplicate values, null values and zeros (more than one time).
- We can differentiate duplicate values by using index.
- It is having an order of insertion by using index.
- It has been included in java 1.2.
- We can also call as sequence.



```
Iterable                    -->allow Homogeneous
     |                      -->allow Hetrogeneous
Collection                  -->Dynamically growable in nature
     |                      -->Order is preserved
List                        -->toString() override
     |-->ArrayList          -->implements Serializable,Cloneable
     |-->LinkedList         -->AL,V--->implements RandomAccess
     |-->Vector             -->AL,V,S-->default size(capacity) is 10
             |-->Stack      -->null allows
                            -->more than one null allow
                            -->duplicate values allows
                            -->No sorting order by default
                            -->     Searching technique by default
                            -->AL loadfactor(fill ratio)--(cc*3/2)+1;
                            -->V,S(loadfactor---cc*2)
```



Methods in List Interface:



```
F:\>javap java.util.List
Compiled from "List.java"
public interface java.util.List extends java.util.Collection{
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean contains(java.lang.Object);
    public abstract java.util.Iterator iterator();
    public abstract java.lang.Object[] toArray();
    public abstract java.lang.Object[] toArray(java.lang.Object[]);
    public abstract boolean add(java.lang.Object);
    public abstract boolean remove(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean addAll(java.util.Collection);
    public abstract boolean addAll(int, java.util.Collection);
    public abstract boolean removeAll(java.util.Collection);
    public abstract boolean retainAll(java.util.Collection);
    public abstract void clear();
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
    public abstract java.lang.Object get(int);
    public abstract java.lang.Object set(int, java.lang.Object);
    public abstract void add(int, java.lang.Object);
    public abstract java.lang.Object remove(int);
    public abstract int indexOf(java.lang.Object);
    public abstract int lastIndexOf(java.lang.Object);
    public abstract java.util.ListIterator listIterator();
    public abstract java.util.ListIterator listIterator(int);
    public abstract java.util.List subList(int, int);
}
```

ArrayList Class:

- It is the child class of List interface.
- ArrayList is a growable and resizable array.
- It allows the duplicate values.
- Heterogeneous (different) objects are allowed.
- Null insertion is possible.
- Insertion order is preserved.
- It is override the toString() method.

Constructors:

(1) ArrayList l=new ArrayList();

Here capacity is empty/null.

The default initial capacity is ten (10).

(2) If the ArrayList reaches its initial capacity, the new ArrayList will be created, with the (current capacity*3/2) +1.

Example:

The minimum capacity is 10. Then after the new capacity is

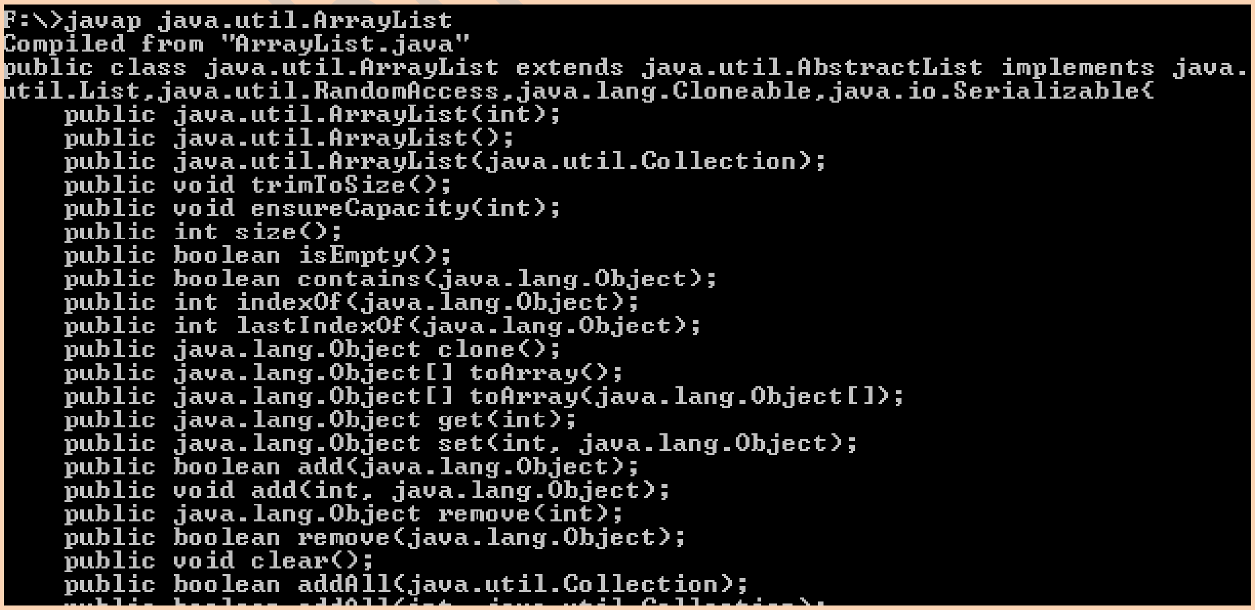
(10*3/2)+1=16.

(3) ArrayList l=new ArrayList(int initial capacity);
(4) ArrayList l=new ArrayList(Collection c)

Here ArrayList will create with the equality size of collection object.

Used at inter conversion between collection objects.

Methods in ArrayList:



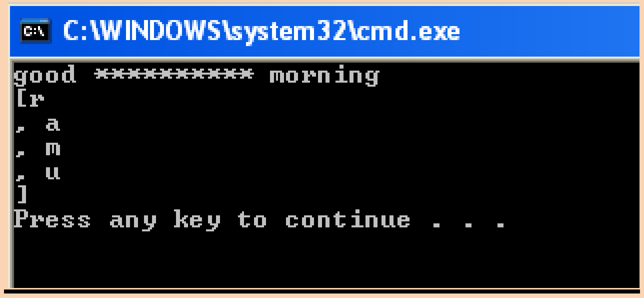
```
F:\>javap java.util.ArrayList
Compiled from "ArrayList.java"
public class java.util.ArrayList extends java.util.AbstractList implements java.
util.List,java.util.RandomAccess,java.lang.Cloneable,java.io.Serializable{
    public java.util.ArrayList(int);
    public java.util.ArrayList();
    public java.util.ArrayList(java.util.Collection);
    public void trimToSize();
    public void ensureCapacity(int);
    public int size();
    public boolean isEmpty();
    public boolean contains(java.lang.Object);
    public int indexOf(java.lang.Object);
    public int lastIndexOf(java.lang.Object);
    public java.lang.Object clone();
    public java.lang.Object[] toArray();
    public java.lang.Object[] toArray(java.lang.Object[]);
    public java.lang.Object get(int);
    public java.lang.Object set(int, java.lang.Object);
    public boolean add(java.lang.Object);
    public void add(int, java.lang.Object);
    public java.lang.Object remove(int);
    public boolean remove(java.lang.Object);
    public void clear();
    public boolean addAll(java.util.Collection);
```

- ArrayList is implements Serializable and Clonable interfaces.
- ArrayList implements the RandomAccess interface.
- Then all the elements in the ArrayList can be retrieved with same speed.
- That's why ArrayList is best suitable for "fast retrieval".
- The drawback of the ArrayList is not suitable for frequent insertion and deletion operations at middle of ArrayList.
- The reason is if we insert or delete the elements at middle of the array the shift operations will be happened, that why it will take more time. This is same for Vector class also.

Program on ArrayList:

```
import java.util.*;

class ArrayListDemo {

        public static void main(String args[]) throws Exception{

        System.out.println("good and bad morning");

        ArrayList l=new ArrayList();

        l.add("r");

        l.add("a");

        l.add("m");

        l.add("u");

        l.add("s");

        l.remove(4);

        System.out.println(l);

                }

}
```

Output:

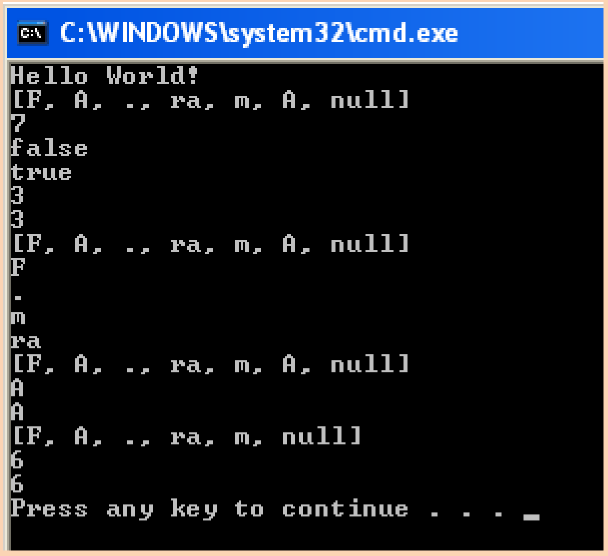


```
good *********** morning
[r
, a
, m
, u
]
Press any key to continue . . .
```

```
import java.util.*;

class ArrayListDemo

{

	public static void main(String[] args)

	{

		System.out.println("Hello World!");

		ArrayList a=new ArrayList();

		a.add("A");

		a.add(".");

		a.add("ra");

		a.add("m");

		a.add(0,"F");

		a.add("A");

		a.add(null);

		System.out.println(a);

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

		System.out.println(a.isEmpty());
```

```
                System.out.println(a.contains("."));
                System.out.println(a.indexOf("ra"));
                System.out.println(a.lastIndexOf("ra"));
                Object o=a.clone();
                System.out.println(o);
                Object o1[]=a.toArray();
                System.out.println(o1[0]);
                System.out.println(o1[2]);
                System.out.println(o1[4]);
                System.out.println(o1[3]);
//              a.clear();
                System.out.println(a);
                System.out.println(a.get(5));
                System.out.println(a.remove(5));
                System.out.println(a);
                a.ensureCapacity(4);
                System.out.println(a.size());
                a.trimToSize();
                System.out.println(a.size());
        }
}
//the default size is 10. and load factor is 3/2+1;
```

Output:



```
Hello World!
[F, A, ., ra, m, A, null]
7
false
true
3
3
[F, A, ., ra, m, A, null]
F
.
m
ra
[F, A, ., ra, m, A, null]
A
A
[F, A, ., ra, m, null]
6
6
Press any key to continue . . .
```

ArrayList is not suitable for frequent insertion and deletion at middle of ArrayList, and then to overcome this problem we have a new class called “LinkedList”.

LinkedList:

- It is the child class of List interface, Deque, Queue.
- LinkedList class extends AbstractSequentialList .
- Duplicate objects will be allowed.
- Null insertion can be allowed.
- Insertion order will be preserved.
- Heterogeneous object can be allowed.
- It is the best suitable for frequent insertion and deletion operations at middle of the list.
- Compare to ArrayList, the LinkedList will not be suitable for only frequent retrieval operation, why because it not implement the RandomAccess interface.
- It is also implement the Serializable and Clonable interfaces.

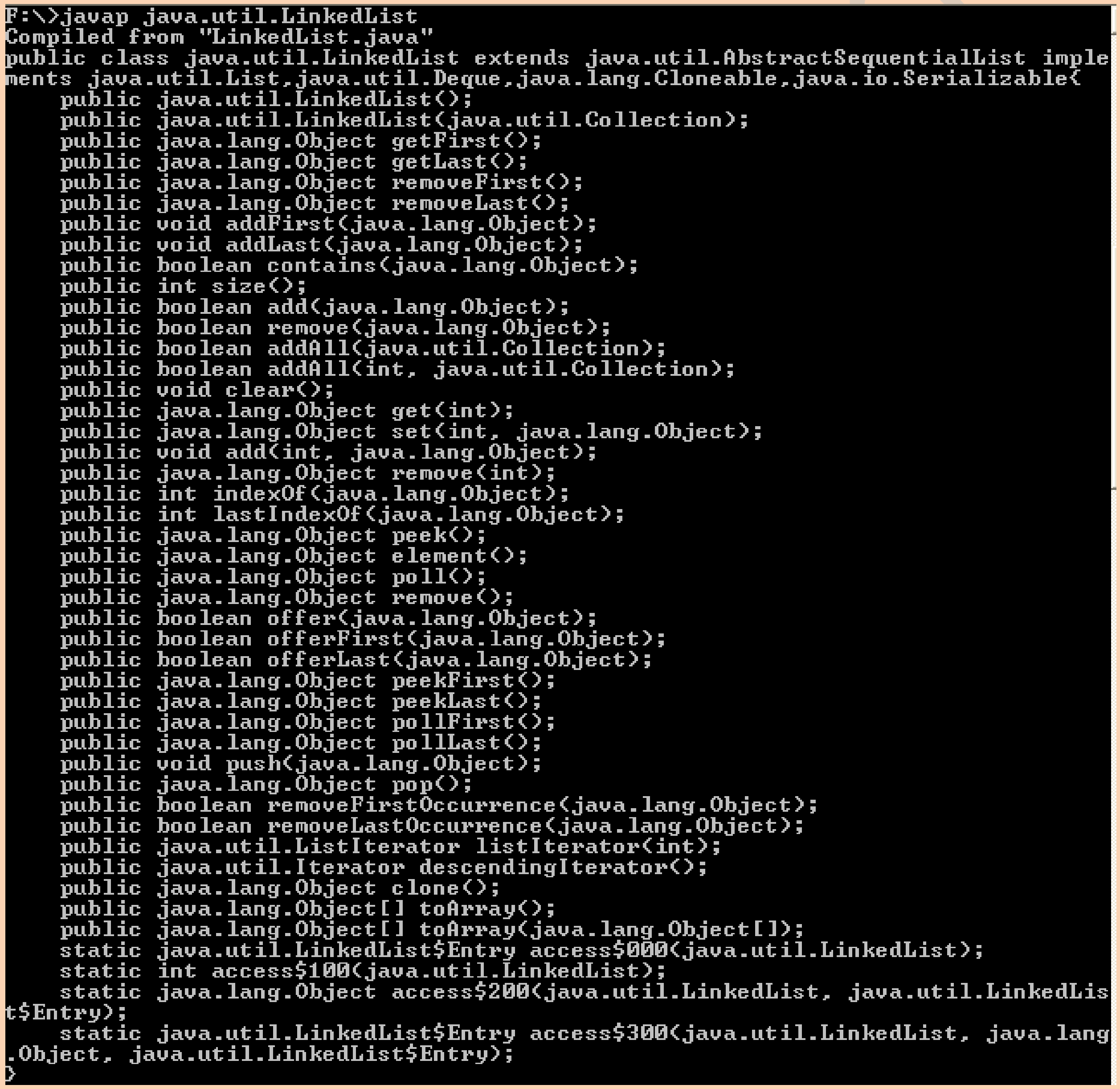
- Up to java1.4 the LinkedList only implements the List interface, whereas in java1.5 LinkedList is also implements the Queue interface.

Constructors in LinkedList:

LinkedList l=new LinkedList();

LinkedList l=new LinkedList(Collection c);

Methods in LinkedList:

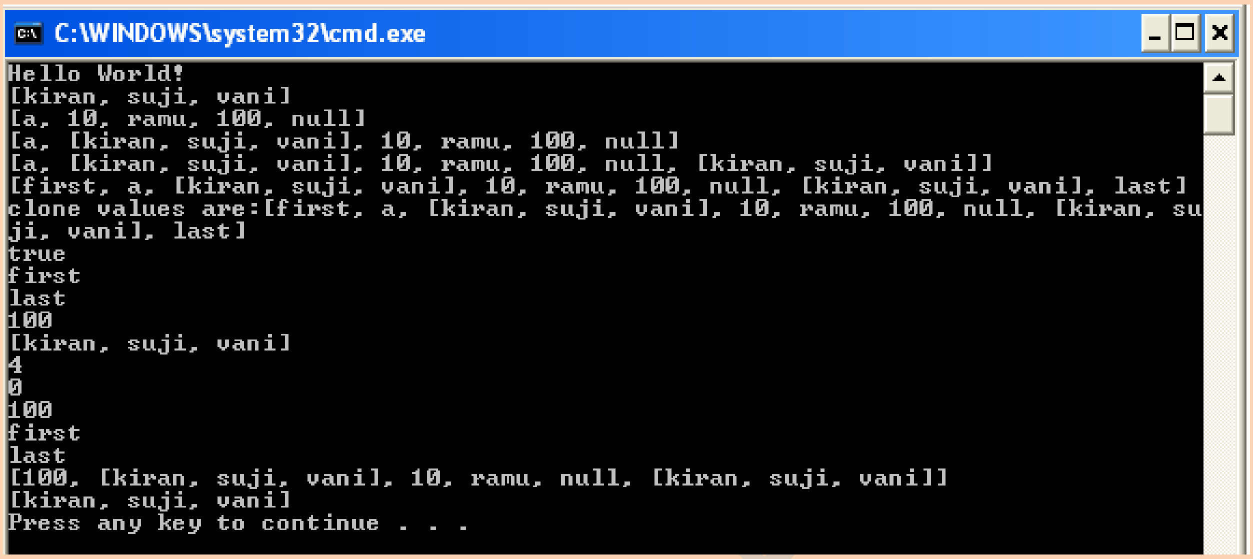


```
F:\>javap java.util.LinkedList
Compiled from "LinkedList.java"
public class java.util.LinkedList extends java.util.AbstractSequentialList imple
ments java.util.List,java.util.Deque,java.lang.Cloneable,java.io.Serializable{
    public java.util.LinkedList();
    public java.util.LinkedList(java.util.Collection);
    public java.lang.Object getFirst();
    public java.lang.Object getLast();
    public java.lang.Object removeFirst();
    public java.lang.Object removeLast();
    public void addFirst(java.lang.Object);
    public void addLast(java.lang.Object);
    public boolean contains(java.lang.Object);
    public int size();
    public boolean add(java.lang.Object);
    public boolean remove(java.lang.Object);
    public boolean addAll(java.util.Collection);
    public boolean addAll(int, java.util.Collection);
    public void clear();
    public java.lang.Object get(int);
    public java.lang.Object set(int, java.lang.Object);
    public void add(int, java.lang.Object);
    public java.lang.Object remove(int);
    public int indexOf(java.lang.Object);
    public int lastIndexOf(java.lang.Object);
    public java.lang.Object peek();
    public java.lang.Object element();
    public java.lang.Object poll();
    public java.lang.Object remove();
    public boolean offer(java.lang.Object);
    public boolean offerFirst(java.lang.Object);
    public boolean offerLast(java.lang.Object);
    public java.lang.Object peekFirst();
    public java.lang.Object peekLast();
    public java.lang.Object pollFirst();
    public java.lang.Object pollLast();
    public void push(java.lang.Object);
    public java.lang.Object pop();
    public boolean removeFirstOccurrence(java.lang.Object);
    public boolean removeLastOccurrence(java.lang.Object);
    public java.util.ListIterator listIterator(int);
    public java.util.Iterator descendingIterator();
    public java.lang.Object clone();
    public java.lang.Object[] toArray();
    public java.lang.Object[] toArray(java.lang.Object[]);
    static java.util.LinkedList$Entry access$000(java.util.LinkedList);
    static int access$100(java.util.LinkedList);
    static java.lang.Object access$200(java.util.LinkedList, java.util.LinkedLis
t$Entry);
    static java.util.LinkedList$Entry access$300(java.util.LinkedList, java.lang
.Object, java.util.LinkedList$Entry);
}
```

```
import java.util.*;
class LinkedListDemo
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
        ArrayList al=new ArrayList();
        al.add("kiran");
        al.add("suji");
        al.add("vani");
        System.out.println(al);
        LinkedList ll=new LinkedList();
        ll.add("a");
        ll.add(10);
        ll.add(new Integer(100));
        ll.add(null);
        ll.add(2,"ramu");
        System.out.println(ll);
        ll.add(1,al);
        System.out.println(ll);
        ll.add(al);
        System.out.println(ll);
```

```
            ll.addFirst("first");
            ll.addLast("last");
            System.out.println(ll);
            Object o=ll.clone();
            System.out.println("clone values are:"+o);
            System.out.println(ll.contains("ramu"));
            System.out.println(ll.getFirst());
            System.out.println(ll.getLast());
            System.out.println(ll.get(5));
            System.out.println(ll.get(2));
            System.out.println(ll.indexOf("ramu"));
            System.out.println(ll.lastIndexOf("first"));
            System.out.println(ll.remove(5));
            System.out.println(ll.removeFirst());
            System.out.println(ll.removeLast());
            ll.set(0,100);
            System.out.println(ll);
            Object o1[]=ll.toArray();
            System.out.println(o1[1]);
            }
    }
```

Output:



```
Hello World!
[kiran, suji, vani]
[a, 10, ramu, 100, null]
[a, [kiran, suji, vani], 10, ramu, 100, null]
[a, [kiran, suji, vani], 10, ramu, 100, null, [kiran, suji, vani]]
[first, a, [kiran, suji, vani], 10, ramu, 100, null, [kiran, suji, vani], last]
clone values are:[first, a, [kiran, suji, vani], 10, ramu, 100, null, [kiran, su
ji, vani], last]
true
first
last
100
[kiran, suji, vani]
4
0
100
first
last
[100, [kiran, suji, vani], 10, ramu, null, [kiran, suji, vani]]
[kiran, suji, vani]
Press any key to continue . . .
```

Vector class:

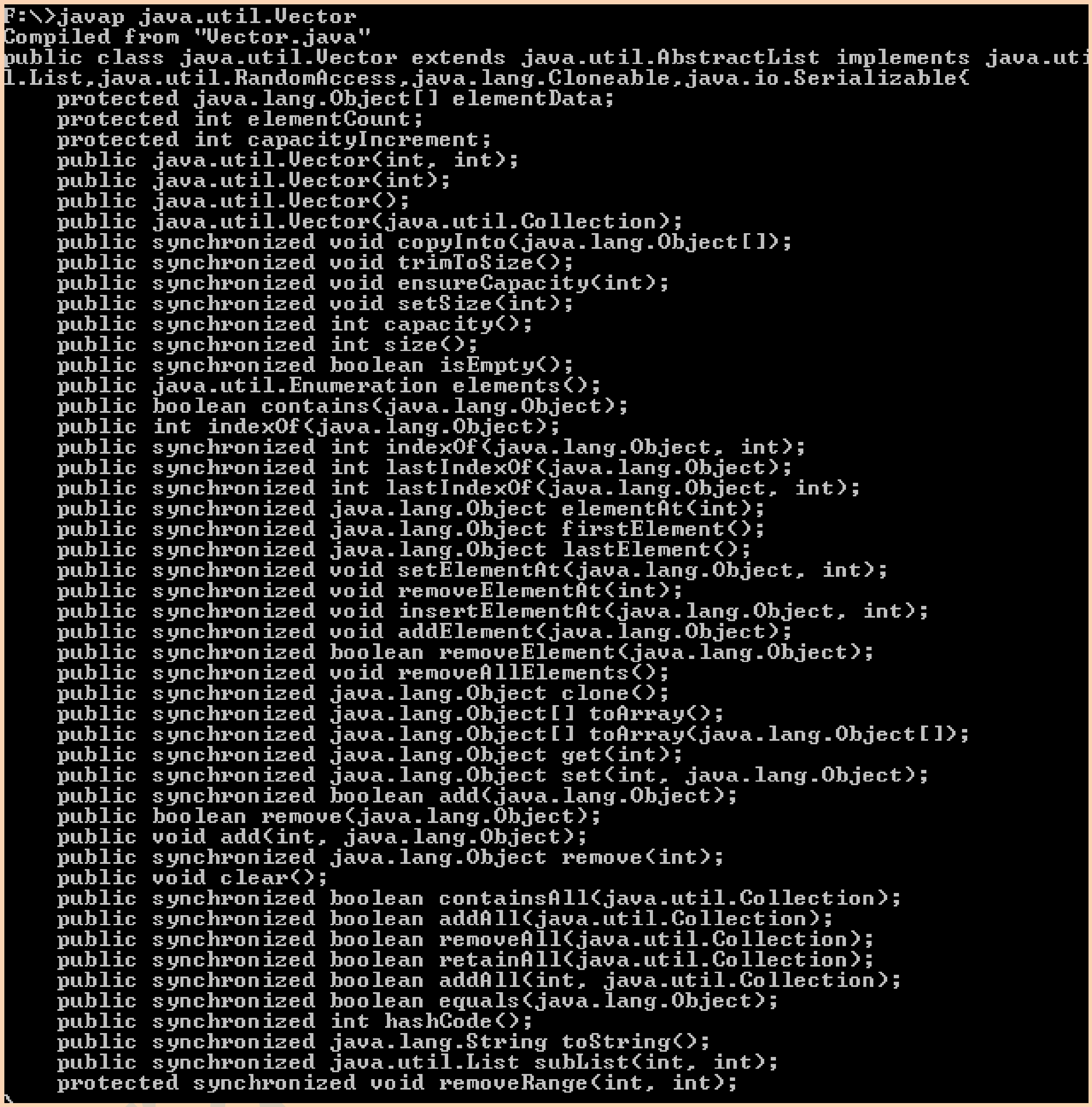
- It is also child class of List interface.
- It allows the duplicate values.
- It is also allows the null values.
- The order must be preserved.
- It has been available from java1.0 onwards.
- Like ArrayList, it is also best suitable for frequent retrieval operations, the reason it can be implements RandomAccess interface.
- Like ArrayList, it is also not suitable for frequent insertion and deletion operations in the middle.
- It is also implements the Clonable, Serializable interface.
- It is somewhat similar to ArrayList but there are some difference between ArrayList and Vector.
- Methods in ArrayList are not synchronized, whereas Vector methods are synchronized.
- ArrayList is not thread safe, whereas Vector is a thread safe.
- Performance is high in ArrayList, where as low in Vector.

- ArrayList by default non-synchronized, where as Vector is synchronized.
- We can get synchronized ArrayList by using following operation. That is
    - Public static List synchronizedList(ArrayList l);

Example:

```
ArrayList          l=new ArrayList();

List      l1= Collections.synchronizedList(l);
```

Methods in Vector:



```
F:\>javap java.util.Vector
Compiled from "Vector.java"
public class java.util.Vector extends java.util.AbstractList implements java.uti
l.List,java.util.RandomAccess,java.lang.Cloneable,java.io.Serializable{
    protected java.lang.Object[] elementData;
    protected int elementCount;
    protected int capacityIncrement;
    public java.util.Vector(int, int);
    public java.util.Vector(int);
    public java.util.Vector();
    public java.util.Vector(java.util.Collection);
    public synchronized void copyInto(java.lang.Object[]);
    public synchronized void trimToSize();
    public synchronized void ensureCapacity(int);
    public synchronized void setSize(int);
    public synchronized int capacity();
    public synchronized int size();
    public synchronized boolean isEmpty();
    public java.util.Enumeration elements();
    public boolean contains(java.lang.Object);
    public int indexOf(java.lang.Object);
    public synchronized int indexOf(java.lang.Object, int);
    public synchronized int lastIndexOf(java.lang.Object);
    public synchronized int lastIndexOf(java.lang.Object, int);
    public synchronized java.lang.Object elementAt(int);
    public synchronized java.lang.Object firstElement();
    public synchronized java.lang.Object lastElement();
    public synchronized void setElementAt(java.lang.Object, int);
    public synchronized void removeElementAt(int);
    public synchronized void insertElementAt(java.lang.Object, int);
    public synchronized void addElement(java.lang.Object);
    public synchronized boolean removeElement(java.lang.Object);
    public synchronized void removeAllElements();
    public synchronized java.lang.Object clone();
    public synchronized java.lang.Object[] toArray();
    public synchronized java.lang.Object[] toArray(java.lang.Object[]);
    public synchronized java.lang.Object get(int);
    public synchronized java.lang.Object set(int, java.lang.Object);
    public synchronized boolean add(java.lang.Object);
    public boolean remove(java.lang.Object);
    public void add(int, java.lang.Object);
    public synchronized java.lang.Object remove(int);
    public void clear();
    public synchronized boolean containsAll(java.util.Collection);
    public synchronized boolean addAll(java.util.Collection);
    public synchronized boolean removeAll(java.util.Collection);
    public synchronized boolean retainAll(java.util.Collection);
    public synchronized boolean addAll(int, java.util.Collection);
    public synchronized boolean equals(java.lang.Object);
    public synchronized int hashCode();
    public synchronized java.lang.String toString();
    public synchronized java.util.List subList(int, int);
    protected synchronized void removeRange(int, int);
```

Constructors in Vector:

(1) Vector v =new Vector()

Initial capacity is 10.

(2) Vector v=new Vector (int x);

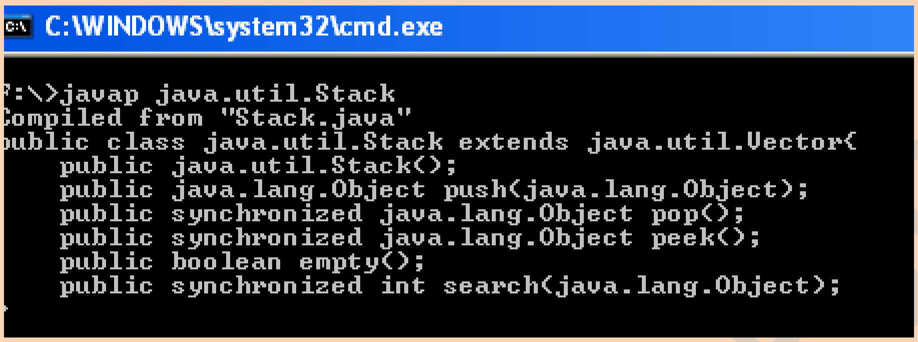
The vector class size will be doubled after reached its max capacity.

(3)Vector v=new Vector (int initial capacity, int increment capacity);

(4)Vector v=new Vector (Collection c)

Stack:

It is the child class of vector.



```
F:\>javap java.util.Stack
Compiled from "Stack.java"
public class java.util.Stack extends java.util.Vector{
    public java.util.Stack();
    public java.lang.Object push(java.lang.Object);
    public synchronized java.lang.Object pop();
    public synchronized java.lang.Object peek();
    public boolean empty();
    public synchronized int search(java.lang.Object);
}
```

Retrieving elements from Collections:

There are four ways to retrieve the element/objects from Collections objects.

They are:

(a) For-each method.
(b) Iterator.
(c) ListIterator.
(d) Enumeration.

The above b, c, d are all interfaces.

For-each loop:

It is a loop like for loop, which executes group of statements for each element of the collection.

Example:

For (variable: collection-object)

{

Statements;

}

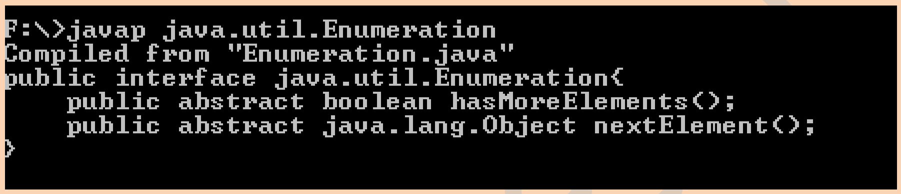
Here collection-object having how many elements, that much of size will be assign into variable and that much of times statements will be executed.

Enumeration:

Enumeration is an interface, which is introduced in java 1.0.

It is useful for retrieving the elements from collection object.

Methods in Enumeration:



```
F:\>javap java.util.Enumeration
Compiled from "Enumeration.java"
public interface java.util.Enumeration{
    public abstract boolean hasMoreElements();
    public abstract java.lang.Object nextElement();
}
```

By using elements () method, we can get Enumeration object.

Elements() method will be available in Vector class in java.util package.

DrawBacks:

- This Enumeration interface doesn't have any remove method. It has only read only methods.
- It is only applicable for Legacy class.
- It is a single direction cursor.

Legacy class:

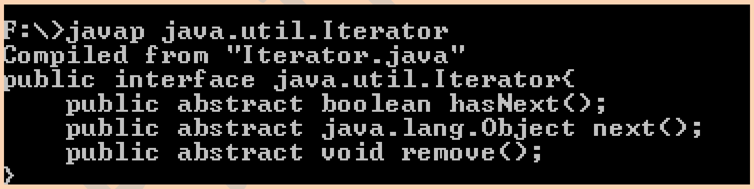
- Legacy classes are those that were built using java 1.1 and java 1.2.
- In general these classes are called as java classes.
- In java 1.0, Vector class was available instead of dynamic array and since there were no ArrayList class people were forced to use Vector for this purpose.
- When java 1.2 was released ArrayList was much more advanced to Vector but has all the features of Vector too.
- So people started using ArrayList instead of Vector.
- And in this case Vector became legacy class.
- Legacy classes are introduced in 1.0 and re-engineered into java 1.2 versions.

Note:

But in general a "legacy" is a term used to define software created using older version of software.

Iterator:

- Iterator is an interface, which was introduced in java 1.2.
- It contains methods to retrieve the elements one by one from a collection object.
- It has 3 methods.



```
F:\>javap java.util.Iterator
Compiled from "Iterator.java"
public interface java.util.Iterator{
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract void remove();
}
```

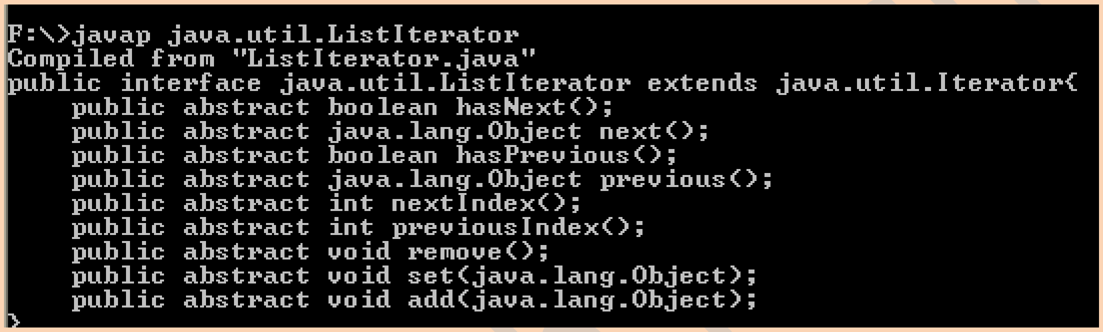
- It will work on any type of classes that is any Collection implemented class.
- By using iterator () method of ArrayList also we can get Iterator object.
- This method will available in List interface.

Drawbacks:

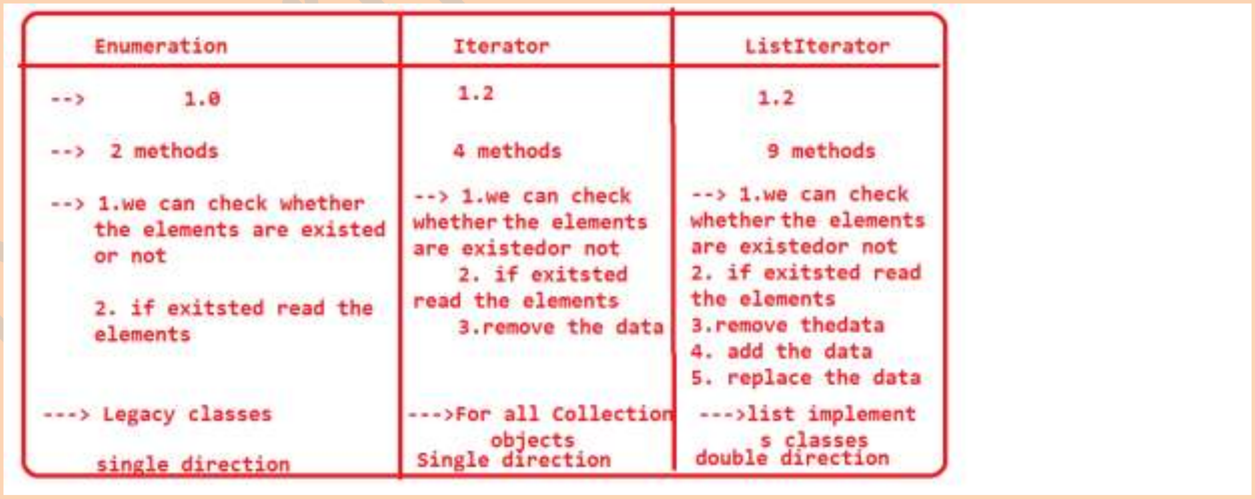
It is also single direction cursor.

It does not have operation like add and replace of new objects.

ListIterator:

- ListItrator is an interface that contains methods to retrieve the elements one by one from a collection object, both in forward and reverse direction.
- It has 9 methods.
- 

```
F:\>javap java.util.ListIterator
Compiled from "ListIterator.java"
public interface java.util.ListIterator extends java.util.Iterator{
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract boolean hasPrevious();
    public abstract java.lang.Object previous();
    public abstract int nextIndex();
    public abstract int previousIndex();
    public abstract void remove();
    public abstract void set(java.lang.Object);
    public abstract void add(java.lang.Object);
}
```

- It is the child interface of Iterator interface.
- It is the bi-directional cursor.
- It has additional feature when compare Iterator and enumeration, that is used for read, remove, replace, add the object.
- 

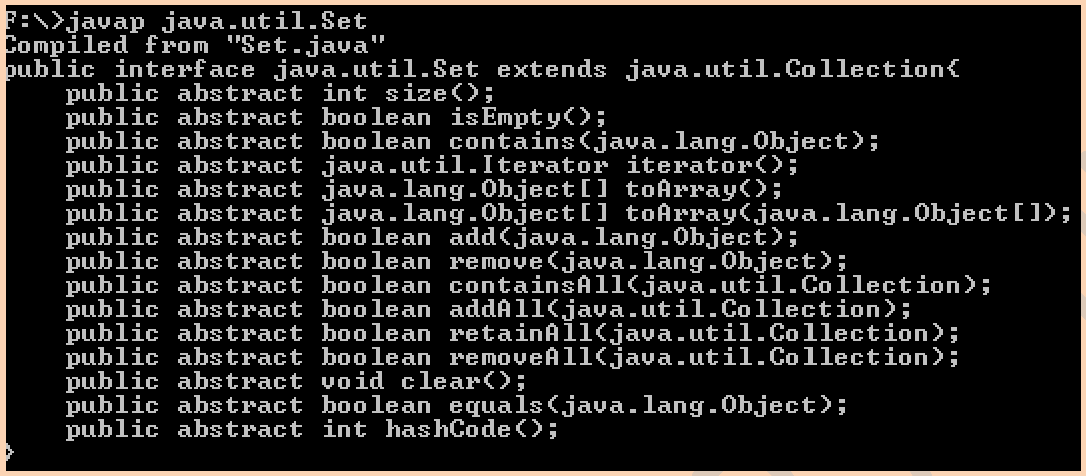
Enumeration	Iterator	ListIterator
--> 1.0	1.2	1.2
--> 2 methods	4 methods	9 methods
--> 1.we can check whether the elements are existed or not 2. if exitsted read the elements	--> 1.we can check whether the elements are existedor not 2. if exitsted read the elements 3.remove the data	--> 1.we can check whether the elements are existedor not 2. if exitsted read the elements 3.remove thedata 4. add the data 5. replace the data
---> Legacy classes single direction	--->For all Collection objects Single direction	--->list implement s classes double direction

Set Interface:

- A Set interface represents a set of elements.

- It doesn’t allow the duplicate elements.
- There is no proper preserve of insertion.

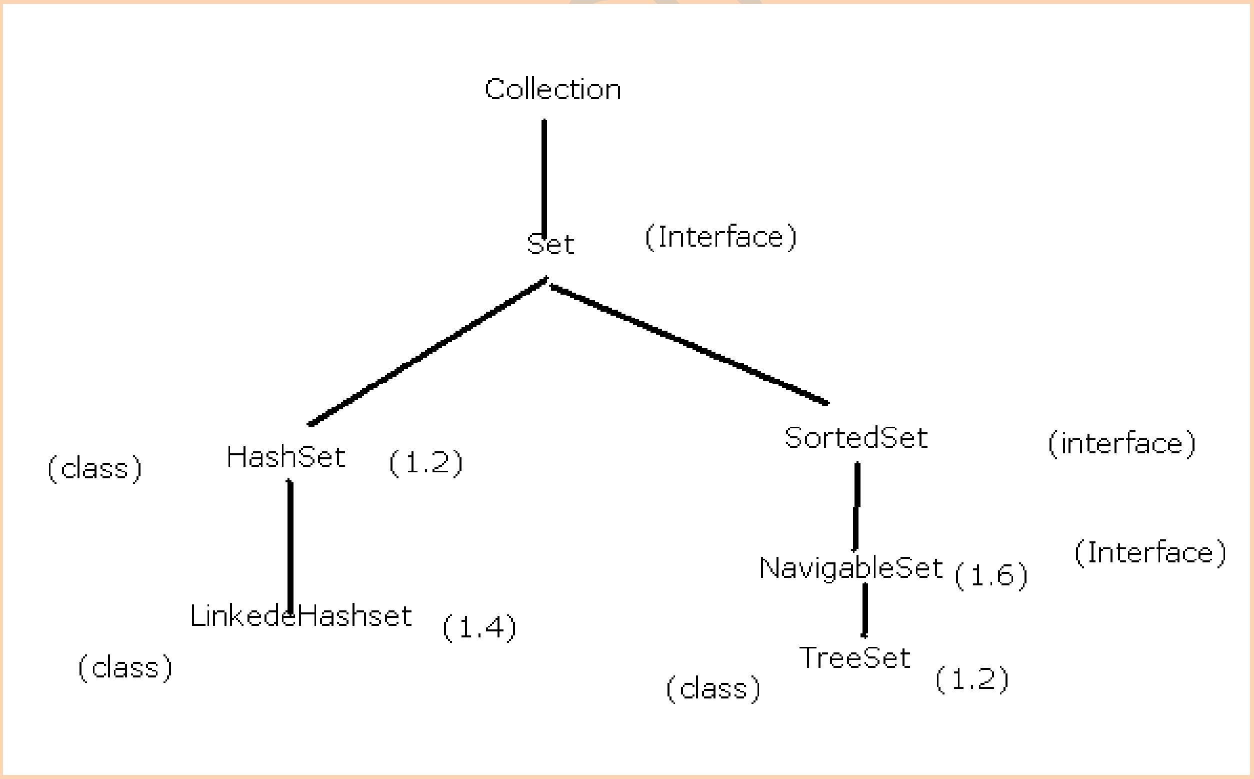
Methods in Set:

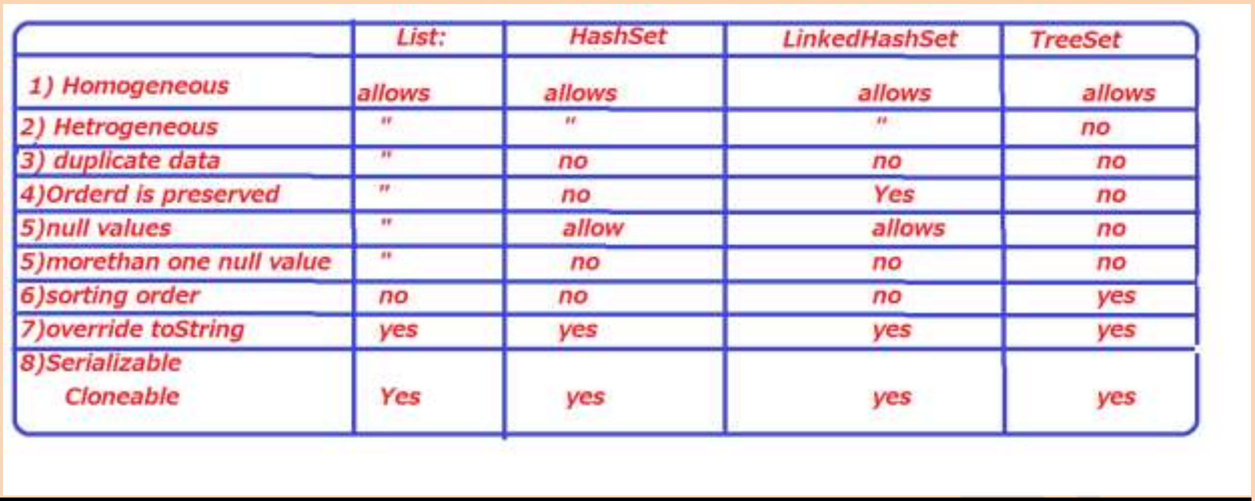


```
F:\>javap java.util.Set
Compiled from "Set.java"
public interface java.util.Set extends java.util.Collection{
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean contains(java.lang.Object);
    public abstract java.util.Iterator iterator();
    public abstract java.lang.Object[] toArray();
    public abstract java.lang.Object[] toArray(java.lang.Object[]);
    public abstract boolean add(java.lang.Object);
    public abstract boolean remove(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean addAll(java.util.Collection);
    public abstract boolean retainAll(java.util.Collection);
    public abstract boolean removeAll(java.util.Collection);
    public abstract void clear();
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
}
```

These methods same as Collection interface methods.

Set interface doesn’t contain any extra methods from Collection.





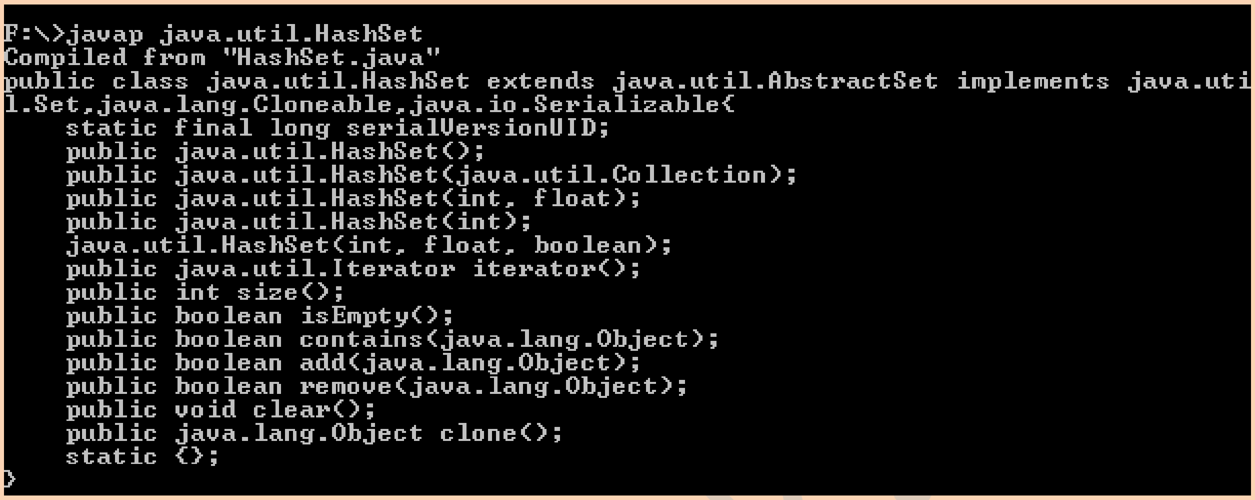
	List:	HashSet	LinkedHashSet	TreeSet
1) Homogeneous	allows	allows	allows	allows
2) Hetrogeneous	"	"	"	no
3) duplicate data	"	no	no	no
4)Orderd is preserved	"	no	Yes	no
5)null values	"	allow	allows	no
5)morethan one null value	"	no	no	no
6)sorting order	no	no	no	yes
7)override toString	yes	yes	yes	yes
8)Serializable Cloneable	Yes	yes	yes	yes

HashSet:

- HashSet is a class, which represents set of elements (objects).
- Insertion order is not preserved.
- Duplication objects will not allowed.
- Heterogeneous objects will be allowed.
- Like List null insertion is possible, but only once.
- It is based on hashCode of an object.
- It is the best suitable for searching operation.
- It is implements Serializable and Clonable interfaces.
- It underlying data structure is HashTable.
- The default inserting order is, if we insert first element there is no specific rule followed by JVM. After that inserting the second element, here JVM will compare the hashCode of first element to hashCode of second element.
- If those two objects hashCode equal then, JVM will use the equals method on those two objects, if the return type true then, the JVM will decide to these two objects are duplicates, it place only one element in the HashSet.
- If equals () method return false then JVM will place these two objects into HashSet.

- If hashCode of the two objects are not equal in the first case, these two objects will place in HashSet.

Methods in HashSet:



```
F:\>javap java.util.HashSet
Compiled from "HashSet.java"
public class java.util.HashSet extends java.util.AbstractSet implements java.uti
l.Set,java.lang.Cloneable,java.io.Serializable{
    static final long serialVersionUID;
    public java.util.HashSet();
    public java.util.HashSet(java.util.Collection);
    public java.util.HashSet(int, float);
    public java.util.HashSet(int);
    java.util.HashSet(int, float, boolean);
    public java.util.Iterator iterator();
    public int size();
    public boolean isEmpty();
    public boolean contains(java.lang.Object);
    public boolean add(java.lang.Object);
    public boolean remove(java.lang.Object);
    public void clear();
    public java.lang.Object clone();
    static {};
}
```

Constructors in HashSet:

(1) HashSet hs=new HashSet();

The default capacity is 16.

Load Factor is 0.75.

(2) HashSet hs=new HashSet(int capacity)

(3) HashSet hs=new HashSet(int initial capacity, float load factory);

Load factor must be in between 0 to 1.

(4) HashSet hs=new HashSet(collection c);

```java
public class CollectionDemo {
    public static void main(String[] args) {
        Integer i = new Integer(10);
        Integer j = new Integer(20);
```

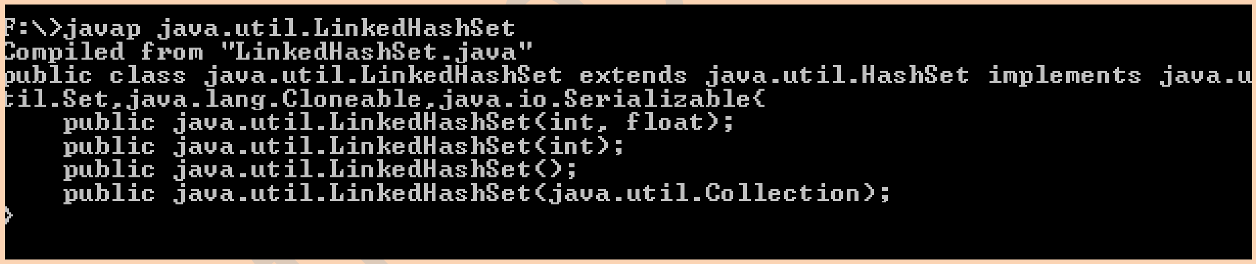
```
            Integer k = new Integer(20);
            HashSet hs = new HashSet();
            hs.add(i);
            hs.add(j);
            hs.add(k);
            System.out.println(hs);

            HashSet hs1 = new HashSet();
            hs1.add("Aa");
            hs1.add("BB");
            System.out.println(hs1);
            System.out.println("Aa".hashCode());
            System.out.println("BB".hashCode());

     }
}
import java.util.Iterator;
import java.util.Scanner;
import java.util.concurrent.CopyOnWriteArraySet;


public class CollectionDemo {
     public static void main(String[] args) {
               /*HashSet hs = new HashSet();
               hs.add(20);
               hs.add(30);
               hs.add(10);
               hs.add(40);
               System.out.println(hs);

               Iterator i = hs.iterator();
               while(i.hasNext()){
                    System.out.println(i.next());
                    hs.add("ram");
               }*/


          CopyOnWriteArraySet cas =  new
CopyOnWriteArraySet();
```

```
            cas.add(20);
            cas.add(30);
            cas.add(10);
            cas.add(40);
            System.out.println(cas);

            Iterator i = cas.iterator();
            while(i.hasNext()){
                System.out.println(i.next());
                //cas.add("ram");
                cas.add(new Scanner(System.in).next());
            }
            System.out.println(cas);
        }
    }
```

LinkedHashSet:

This is the sub class of HashSet.

It doesn’t have any additional methods.



```
F:\>javap java.util.LinkedHashSet
Compiled from "LinkedHashSet.java"
public class java.util.LinkedHashSet extends java.util.HashSet implements java.u
til.Set,java.lang.Cloneable,java.io.Serializable{
    public java.util.LinkedHashSet(int, float);
    public java.util.LinkedHashSet(int);
    public java.util.LinkedHashSet();
    public java.util.LinkedHashSet(java.util.Collection);
}
```

Features:

Introduced in java 1.4.

Underlying data structure is LinkedList + HashTable. That’s why it maintains the insertion order.

It doesn’t allow the duplicate values that are why it is best suited for cache memory application.

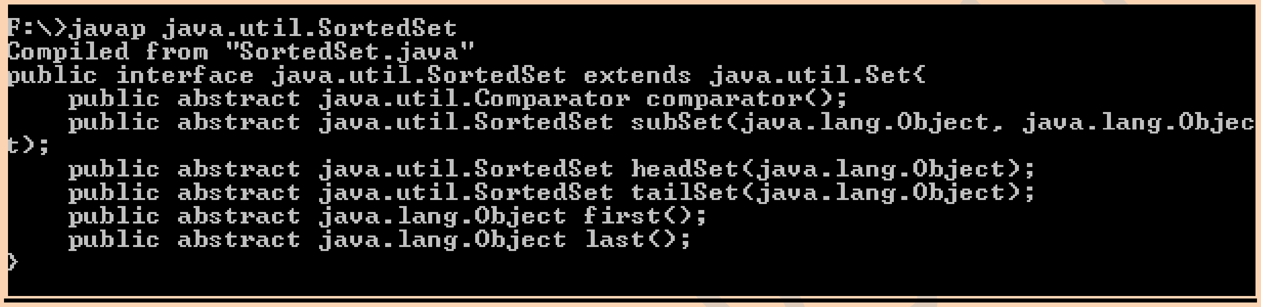
SortedSet:

SortedSet is a child interface of Set interface.

Its name specifies that, it will useful for grouping the unique object according to some sorting order.

The sorting order is may be default or customized sorting order.

Methods in SortedSet:



```
F:\>javap java.util.SortedSet
Compiled from "SortedSet.java"
public interface java.util.SortedSet extends java.util.Set{
    public abstract java.util.Comparator comparator();
    public abstract java.util.SortedSet subSet(java.lang.Object, java.lang.Objec
t);
    public abstract java.util.SortedSet headSet(java.lang.Object);
    public abstract java.util.SortedSet tailSet(java.lang.Object);
    public abstract java.lang.Object first();
    public abstract java.lang.Object last();
}
```

comparator():

This method will return Comparator interface.

If the sorting technique is default, then this method returns null.

TreeSet:

- The underlying data structure is balanced tree.
- Duplicate values are not allowed.
- Insertion order is preserved.
- Heterogeneous values are not allowed.

Constructors in TreeSet:

(1) TreeSet ts=new TreeSet().

Creates an empty treeset object, where the sorting order is default natural sorting order.

The elements which are inserted into the set must implements Comparable interface.

(2) TreeSet ts=new TreeSet(Comparator obj)

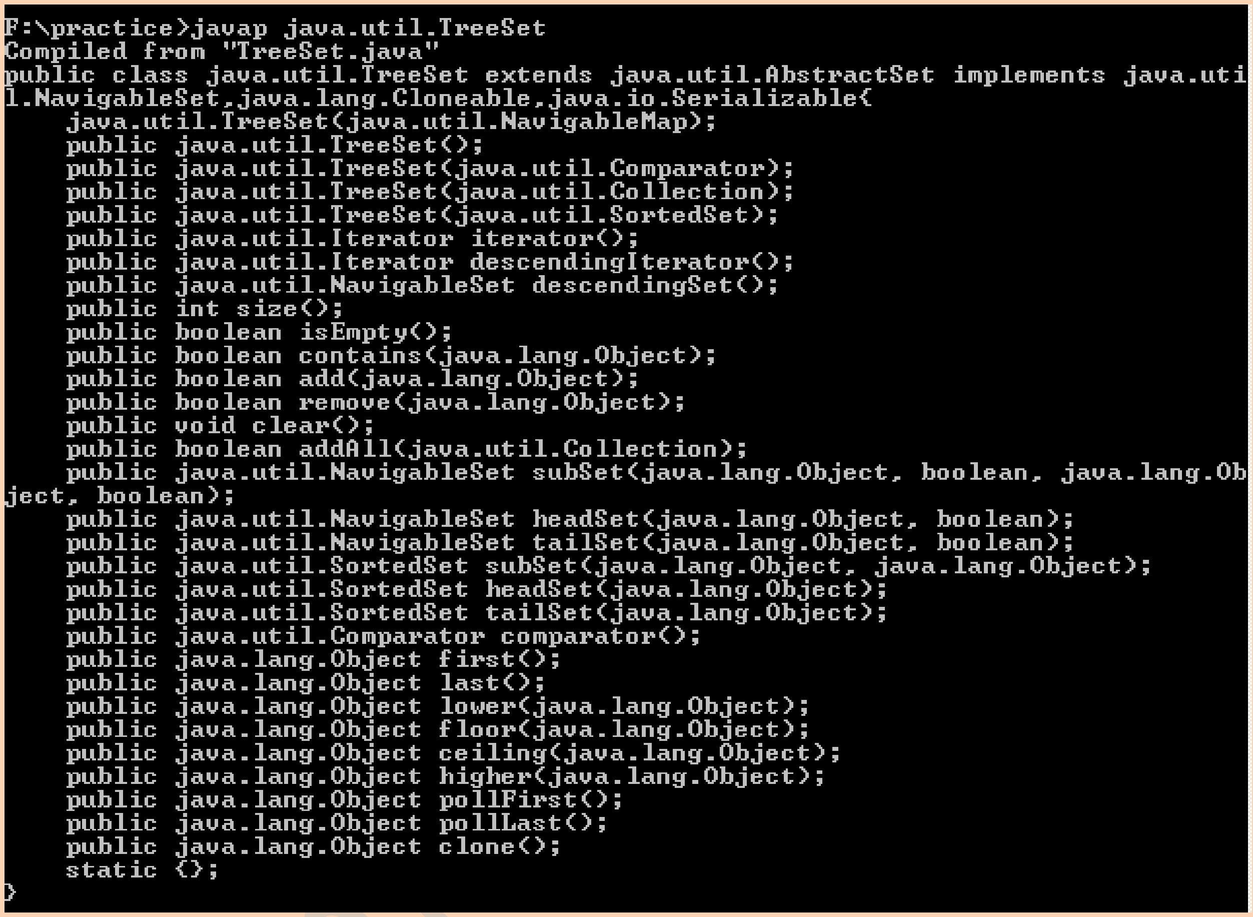
Creates empty TreeSet object, where sorting order is specific by Comparator.

This is customized sorting order.

(3) TreeSet ts=new TreeSet(Collection c)
(4) TreeSet ts=new TreeSet(SortedSet s)

- Creates a new tree set containing the same elements and using the same ordering as the specified SortedSet.
- For empty TreeSet as the first value null insertion is possible, but after inserting null value, if are trying insert any other null value then we will get NullPointerException.
- Whereas non-empty, if we are inserting null value, then we will get NullPointerException.
- If we want depend on natural sorting order, then we should place the homogeneous objects and comparable otherwise we will get ClassCastException.
- If an object is a comparable, if and only if the corresponding class will implements Comparable interface.
- String and all wrapper classes implemented Comparable interface.

- Whereas StrinBuffer, StringBuilder does not implement the Comparable interface.



```
F:\practice>javap java.util.TreeSet
Compiled from "TreeSet.java"
public class java.util.TreeSet extends java.util.AbstractSet implements java.uti
l.NavigableSet,java.lang.Cloneable,java.io.Serializable{
    java.util.TreeSet(java.util.NavigableMap);
    public java.util.TreeSet();
    public java.util.TreeSet(java.util.Comparator);
    public java.util.TreeSet(java.util.Collection);
    public java.util.TreeSet(java.util.SortedSet);
    public java.util.Iterator iterator();
    public java.util.Iterator descendingIterator();
    public java.util.NavigableSet descendingSet();
    public int size();
    public boolean isEmpty();
    public boolean contains(java.lang.Object);
    public boolean add(java.lang.Object);
    public boolean remove(java.lang.Object);
    public void clear();
    public boolean addAll(java.util.Collection);
    public java.util.NavigableSet subSet(java.lang.Object, boolean, java.lang.Ob
ject, boolean);
    public java.util.NavigableSet headSet(java.lang.Object, boolean);
    public java.util.NavigableSet tailSet(java.lang.Object, boolean);
    public java.util.SortedSet subSet(java.lang.Object, java.lang.Object);
    public java.util.SortedSet headSet(java.lang.Object);
    public java.util.SortedSet tailSet(java.lang.Object);
    public java.util.Comparator comparator();
    public java.lang.Object first();
    public java.lang.Object last();
    public java.lang.Object lower(java.lang.Object);
    public java.lang.Object floor(java.lang.Object);
    public java.lang.Object ceiling(java.lang.Object);
    public java.lang.Object higher(java.lang.Object);
    public java.lang.Object pollFirst();
    public java.lang.Object pollLast();
    public java.lang.Object clone();
    static {};
}
```

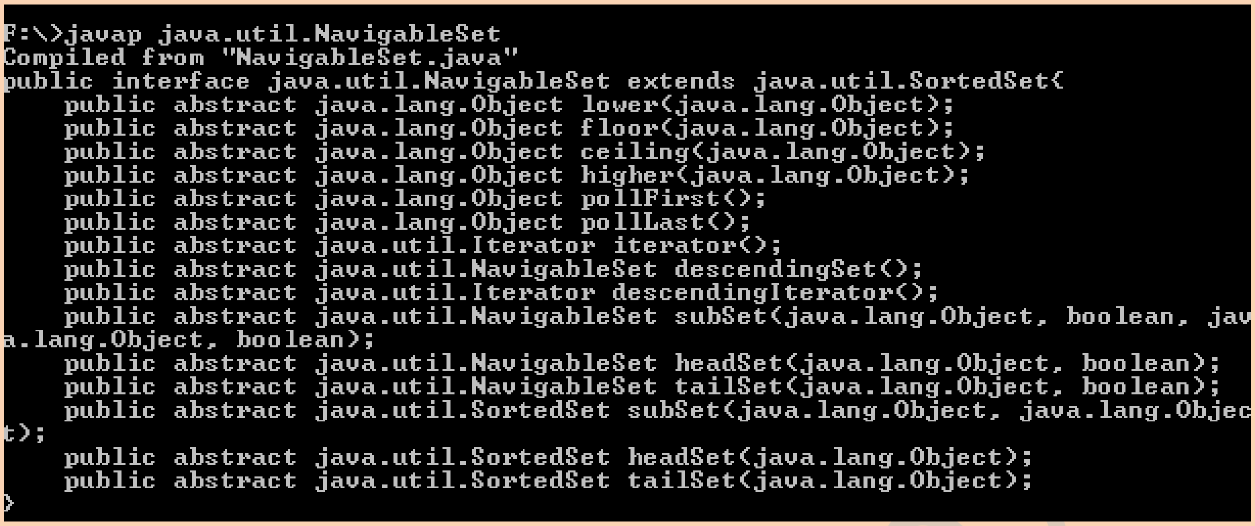
<u>NavigableSet:</u>

For navigation of SortedSet purpose, the sun micro people introduce a new concept called NavigableSet.

It is the child interface of SortedSet interface.

It has some method for navigation purpose.

<u>Methods in NavigableSet:</u>



```
F:\>javap java.util.NavigableSet
Compiled from "NavigableSet.java"
public interface java.util.NavigableSet extends java.util.SortedSet{
    public abstract java.lang.Object lower(java.lang.Object);
    public abstract java.lang.Object floor(java.lang.Object);
    public abstract java.lang.Object ceiling(java.lang.Object);
    public abstract java.lang.Object higher(java.lang.Object);
    public abstract java.lang.Object pollFirst();
    public abstract java.lang.Object pollLast();
    public abstract java.util.Iterator iterator();
    public abstract java.util.NavigableSet descendingSet();
    public abstract java.util.Iterator descendingIterator();
    public abstract java.util.NavigableSet subSet(java.lang.Object, boolean, jav
a.lang.Object, boolean);
    public abstract java.util.NavigableSet headSet(java.lang.Object, boolean);
    public abstract java.util.NavigableSet tailSet(java.lang.Object, boolean);
    public abstract java.util.SortedSet subSet(java.lang.Object, java.lang.Objec
t);
    public abstract java.util.SortedSet headSet(java.lang.Object);
    public abstract java.util.SortedSet tailSet(java.lang.Object);
}
```

Comparable Interface:

It is available in java.lang package and contains only one method. That is

Public abstract int compareTo(java.lang.Object obj);

Obj1.compareTo(obj2)

Rules:

(1) Return positive number if obj1 comes before obj2.
(2) Return negative number if obj1 comes after obj2.
(3) Return zero, if obj1 and obj2 are equal.

When we inserted elements into TreeSet, if we are not using any sorting order, then we should call compareTo() method and default natural sorting order.

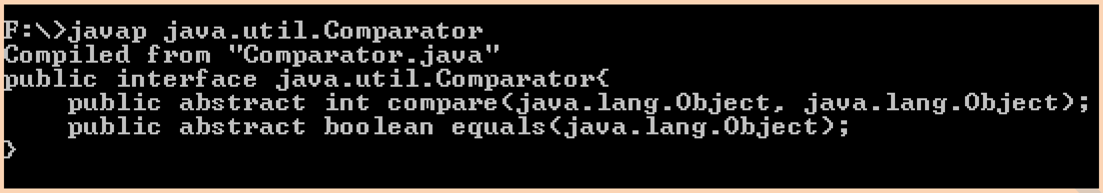
Comparator interface:

If we want inserted elements into TreeSet with our own or customized sorting technique then we should go for Comparator interface.

This interface will be available in java.util package.

Methods in Comparator:

It has two methods.



```
F:\>javap java.util.Comparator
Compiled from "Comparator.java"
public interface java.util.Comparator{
    public abstract int compare(java.lang.Object, java.lang.Object);
    public abstract boolean equals(java.lang.Object);
}
```

(1)Public abstract in compare(obj1,obj2):

Rules:

(1) Return positive number if obj1 comes before obj2.
(2) Return negative number if obj1 comes after obj2.
(3) Return zero, if obj1 and obj2 are equal.

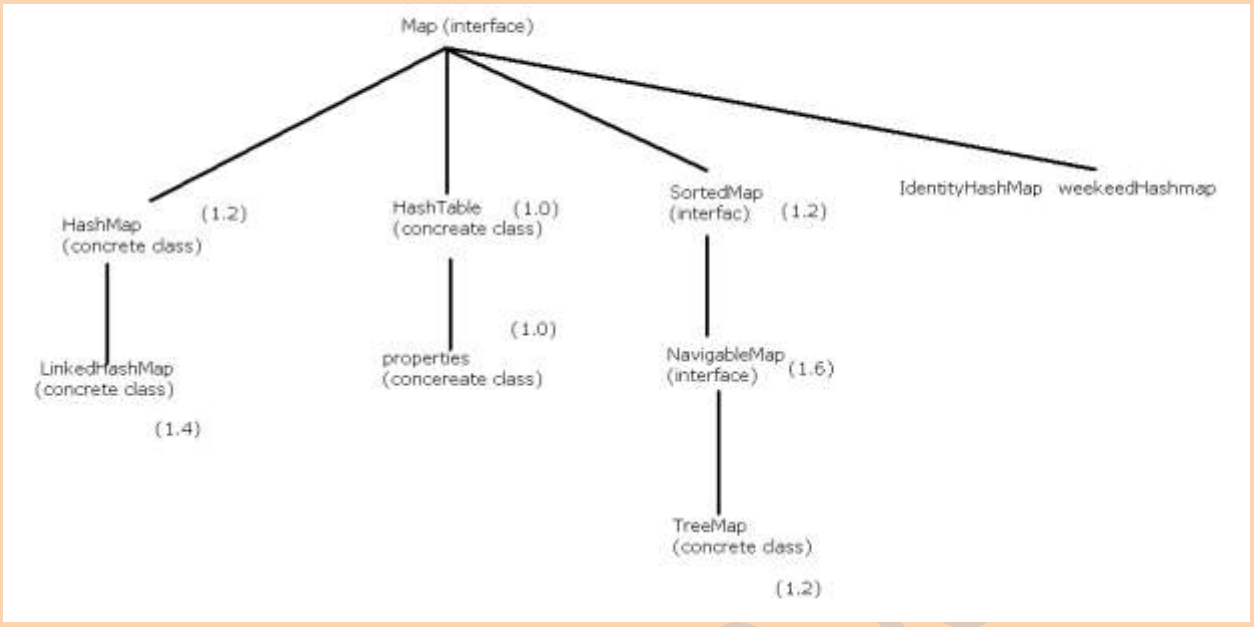
(2)public abstract boolean equals(Obj);

- If we using Comparator interface, we would provide implementation of compare(ob1,obj2).
- Implementing equals() is optional, because it has implementation in Object class.
- We can provide heterogeneous objects also.

Map Hierarchy:

Map hierarchy classes are used to collect elements in (key, value) pair of format.

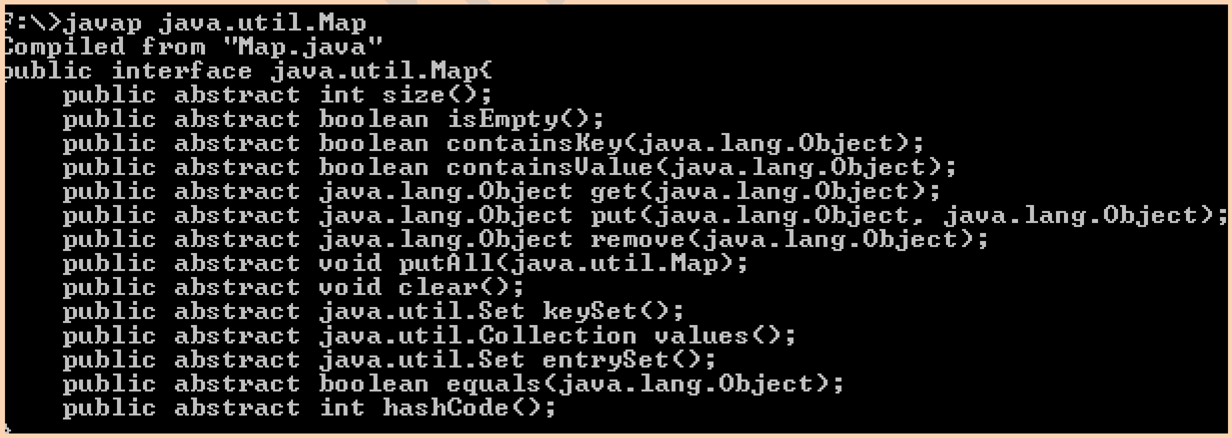
Both key and value are objects only.


Map (interface)
HashMap (concrete class) (1.2)
HashTable (concreate class) (1.0)
SortedMap (interfac) (1.2)
IdentityHashMap
weekeedHashmap
LinkedHashMap (concrete class) (1.4)
properties (concereate class) (1.0)
NavigableMap (interface) (1.6)
TreeMap (concrete class) (1.2)

Map Interface:

- Ina Map interface key should be unique, whereas value can be duplicate.
- The pair of (key, value) is called one entry.
- Map interface is not a child interface of Collection interface.

Methods in Map:

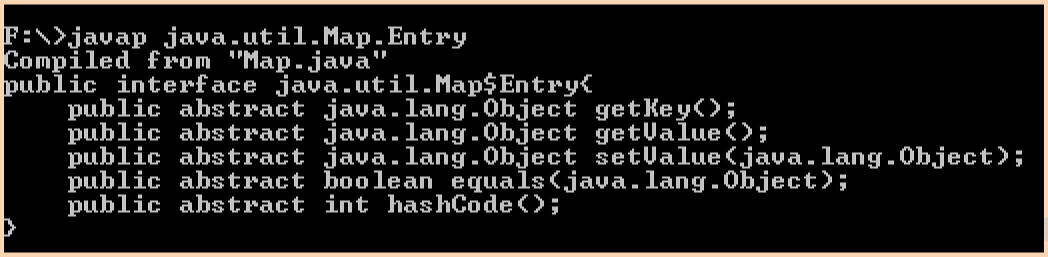


```
F:\>javap java.util.Map
Compiled from "Map.java"
public interface java.util.Map{
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean containsKey(java.lang.Object);
    public abstract boolean containsValue(java.lang.Object);
    public abstract java.lang.Object get(java.lang.Object);
    public abstract java.lang.Object put(java.lang.Object, java.lang.Object);
    public abstract java.lang.Object remove(java.lang.Object);
    public abstract void putAll(java.util.Map);
    public abstract void clear();
    public abstract java.util.Set keySet();
    public abstract java.util.Collection values();
    public abstract java.util.Set entrySet();
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
```

Entry Interface:

A map is a collection of entries (key, value), hence Entry is an inner interface defined in Map interface.

It has 5 methods.

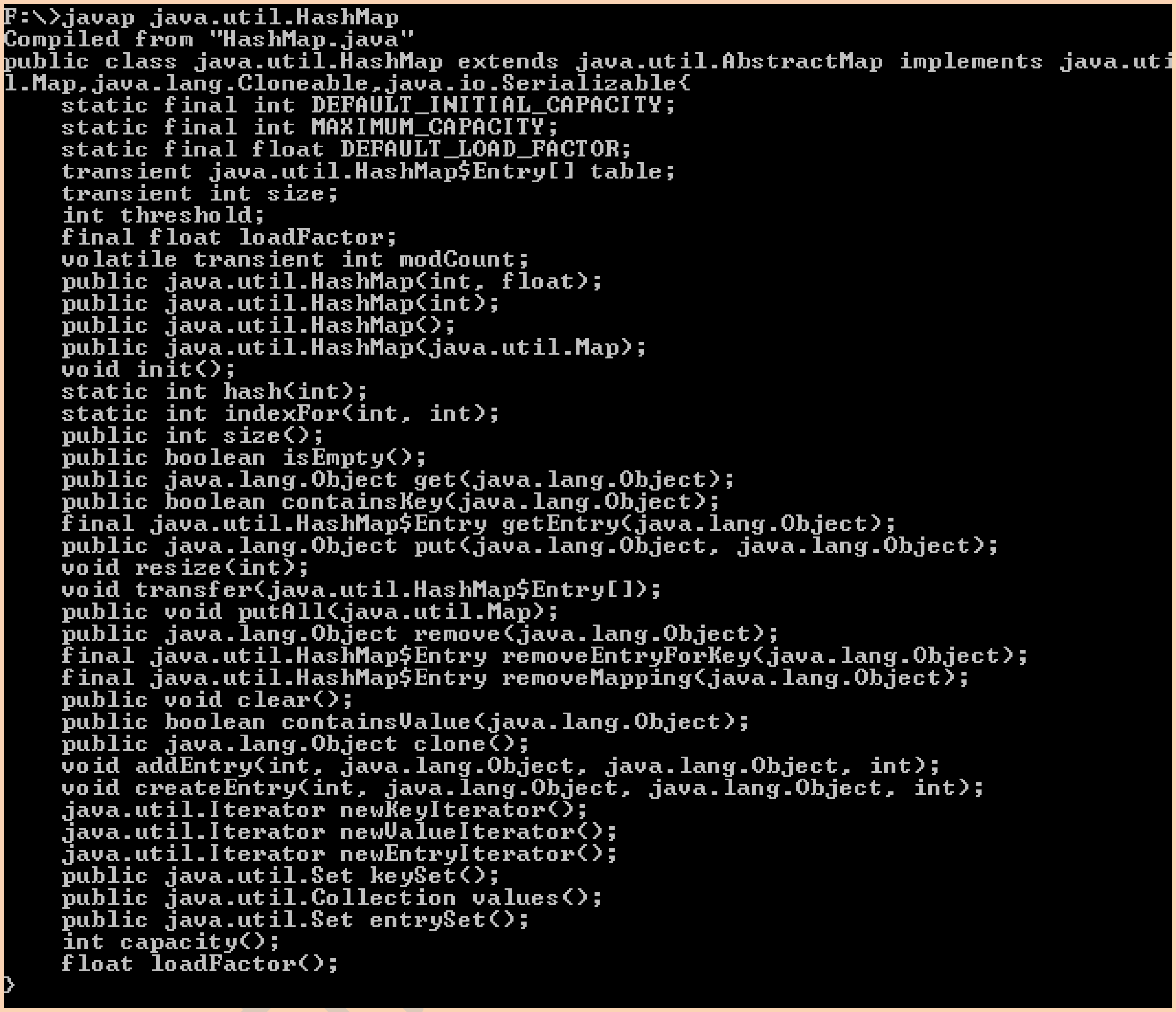


```
F:\>javap java.util.Map.Entry
Compiled from "Map.java"
public interface java.util.Map$Entry{
    public abstract java.lang.Object getKey();
    public abstract java.lang.Object getValue();
    public abstract java.lang.Object setValue(java.lang.Object);
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
}
```

HashMap class:

- HashMap is a collection that stores elements in the form of key-value pairs.
- If key is provided later, its corresponding value can be easily retrieved from the HashMap.
- Keys should be unique.
- Value may or may be duplicate.
- The underlying data structure is Hashtable.
- It not a synchronized, if multiple threads work on this object, we might be get unreliable values.
- Insertion order is not preserved.
- Heterogeneous object can be allowed.
- It has been based on hashCode of the keys.
- Null insertion can be possible for both key and values.
- It has been introduced in java 1.2.
- HashMap by default it is not a synchronized, but make it as synchronized(thread safe).
- By using Collection.synchronziedMap(java.util.Map object) we make it as synchronized.

Methods in HashMap:



```
F:\>javap java.util.HashMap
Compiled from "HashMap.java"
public class java.util.HashMap extends java.util.AbstractMap implements java.uti
l.Map,java.lang.Cloneable,java.io.Serializable{
    static final int DEFAULT_INITIAL_CAPACITY;
    static final int MAXIMUM_CAPACITY;
    static final float DEFAULT_LOAD_FACTOR;
    transient java.util.HashMap$Entry[] table;
    transient int size;
    int threshold;
    final float loadFactor;
    volatile transient int modCount;
    public java.util.HashMap(int, float);
    public java.util.HashMap(int);
    public java.util.HashMap();
    public java.util.HashMap(java.util.Map);
    void init();
    static int hash(int);
    static int indexFor(int, int);
    public int size();
    public boolean isEmpty();
    public java.lang.Object get(java.lang.Object);
    public boolean containsKey(java.lang.Object);
    final java.util.HashMap$Entry getEntry(java.lang.Object);
    public java.lang.Object put(java.lang.Object, java.lang.Object);
    void resize(int);
    void transfer(java.util.HashMap$Entry[]);
    public void putAll(java.util.Map);
    public java.lang.Object remove(java.lang.Object);
    final java.util.HashMap$Entry removeEntryForKey(java.lang.Object);
    final java.util.HashMap$Entry removeMapping(java.lang.Object);
    public void clear();
    public boolean containsValue(java.lang.Object);
    public java.lang.Object clone();
    void addEntry(int, java.lang.Object, java.lang.Object, int);
    void createEntry(int, java.lang.Object, java.lang.Object, int);
    java.util.Iterator newKeyIterator();
    java.util.Iterator newValueIterator();
    java.util.Iterator newEntryIterator();
    public java.util.Set keySet();
    public java.util.Collection values();
    public java.util.Set entrySet();
    int capacity();
    float loadFactor();
}
```

Constructors in HashMap:

(1) HashMap h = new HashMap()

Creates an empty HashMap object with default initial capacity is 16. Default fill ratio 0.75,like HashSet.

(2) HashMap h=new HashMap(int initial capacity)

(3)HashMap h=new HashMap(int initial capacity, float fillratio).

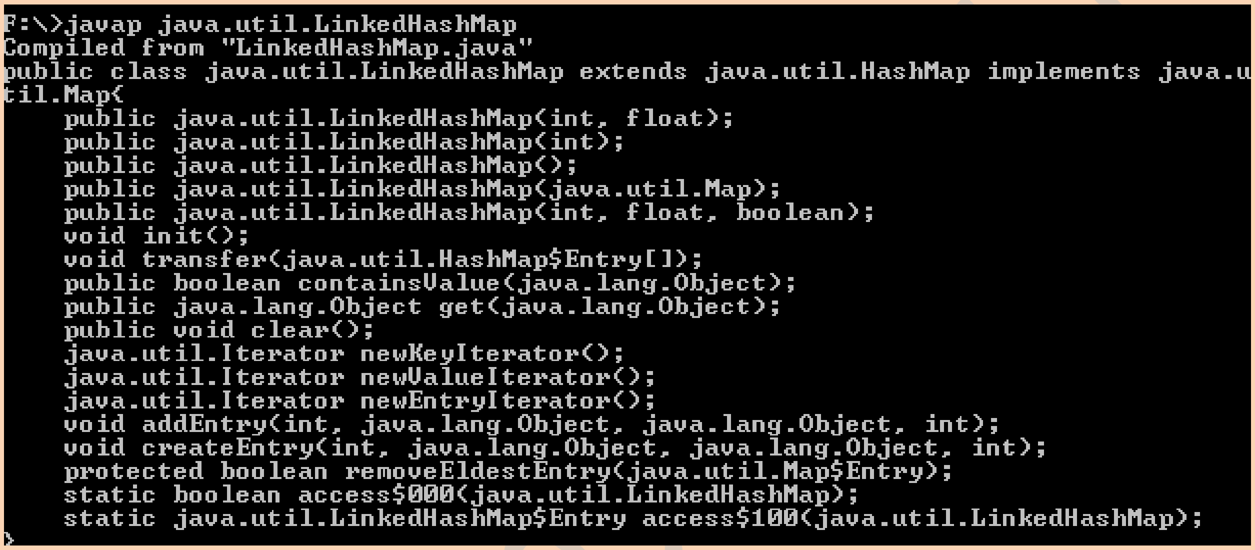
(4) HashMap h=new HashMap(Map m);

LinkedHashMap:

- This is similar to HashMap.
- The underlying data structure is HashTable+LinkedList.
- Insertion order is preserved.
- Introduced in java 1.2.

```
package collection;

import java.util.Iterator;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

public class CollectionDemo {
    public static void main(String[] args) {
        /*HashMap hm = new HashMap();
        hm.put(101,"ram");
        hm.put(102,"sam");

        System.out.println(hm);
        Set s = hm.keySet();
        System.out.println(s);
        Iterator i = s.iterator();
        while(i.hasNext()){
            System.out.println(i.next());
            hm.put(111,222);
        }*/
        ConcurrentHashMap hm = new ConcurrentHashMap();
        hm.put(101,"ram");
        hm.put(102,"sam");
        System.out.println(hm);
        Set s = hm.keySet();
        System.out.println(s);
        Iterator i = s.iterator();
        while(i.hasNext()){
            System.out.println(i.next());
            hm.put(111,222);
        }
        System.out.println(hm);
    }
}
```

Methods in LinkedHashMap:



```
F:\>javap java.util.LinkedHashMap
Compiled from "LinkedHashMap.java"
public class java.util.LinkedHashMap extends java.util.HashMap implements java.u
til.Map{
    public java.util.LinkedHashMap(int, float);
    public java.util.LinkedHashMap(int);
    public java.util.LinkedHashMap();
    public java.util.LinkedHashMap(java.util.Map);
    public java.util.LinkedHashMap(int, float, boolean);
    void init();
    void transfer(java.util.HashMap$Entry[]);
    public boolean containsValue(java.lang.Object);
    public java.lang.Object get(java.lang.Object);
    public void clear();
    java.util.Iterator newKeyIterator();
    java.util.Iterator newValueIterator();
    java.util.Iterator newEntryIterator();
    void addEntry(int, java.lang.Object, java.lang.Object, int);
    void createEntry(int, java.lang.Object, java.lang.Object, int);
    protected boolean removeEldestEntry(java.util.Map$Entry);
    static boolean access$000(java.util.LinkedHashMap);
    static java.util.LinkedHashMap$Entry access$100(java.util.LinkedHashMap);
}
```

Hashtable:

- Hashtable is similar to HashMap, which holds elements in the form of key-value pairs.
- It is a synchronized class.
- It is an thread safe.
- Performance is low, when compare to HashMap.
- Null value cannot be allowed in the place of key and value.
- It has introduced in java 1.0.
- The underlying data structure is Hashtable.
- Insertion order is not preserved and it is based on hashCode of an objects.
- Duplicate keys will not be allowed, but duplicate values can be allowed.

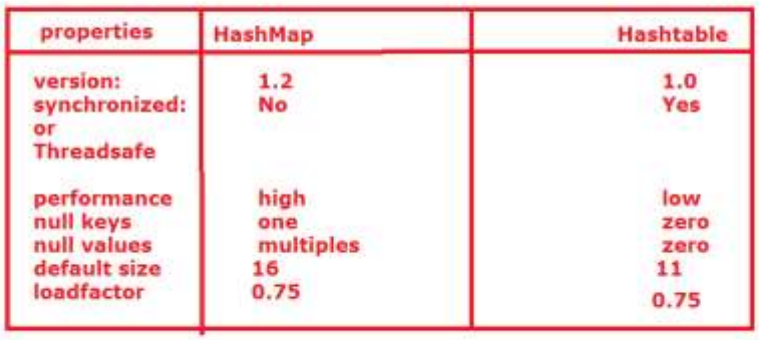
- Heterogeneous objects will be allowed for both key and values.

Constructors in Hashtable:

(1) Hashtable ht=new Hashtable();

Initial capacity is 11 and default fillratio 0.75.

(2) Hashtable ht=new Hashtable(int initialcapacity);
(3) Hashtable ht=new Hashtable(int initialcapacity, float fillratrio);
(4) Hashtable ht=new Hashtable(map m);



properties	HashMap	Hashtable
version:	1.2	1.0
synchronized: or Threadsafe	No	Yes
performance	high	low
null keys	one	zero
null values	multiples	zero
default size	16	11
loadfactor	0.75	0.75

IdentityHashMap:

In the case of duplication values in IdentityHashMap, the JVM will be uses the "==" operator for identifies the duplicates.

Where as in HashMap, the JVM will uses equals() method to identifies the duplicates.

WeakHashMap:

- It is exactly similar to HashMap.
- HashMap dominates GarbageCollector that is if any object associated with HashMap it is not eligible for Garbage Collector even though it doesn't have external references.

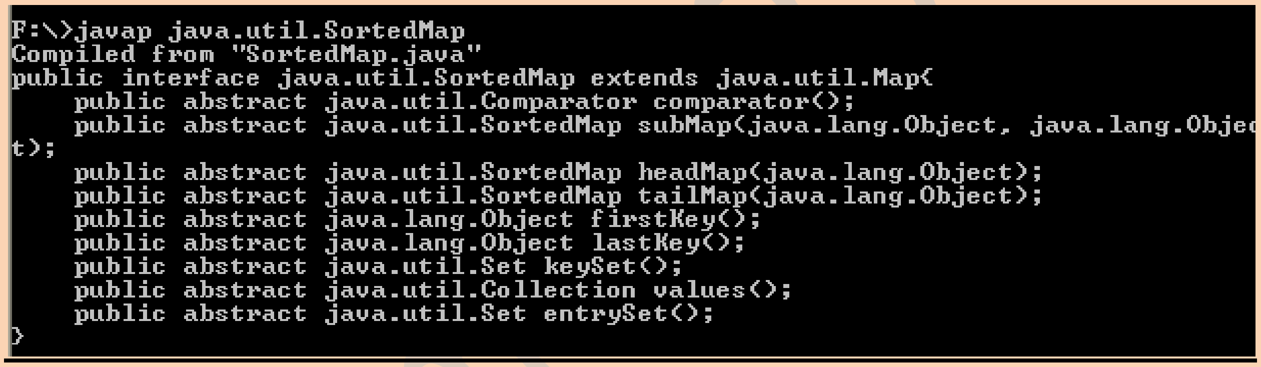
- But in the case of WeakedHashMap an object is eligible for Garbage Collector.

SortedMap interface:

It is child interface of the Map.

If we want represents all the entries according to default sorting order of keys, then we should go for SortedMap.

Methods in SortedMap:



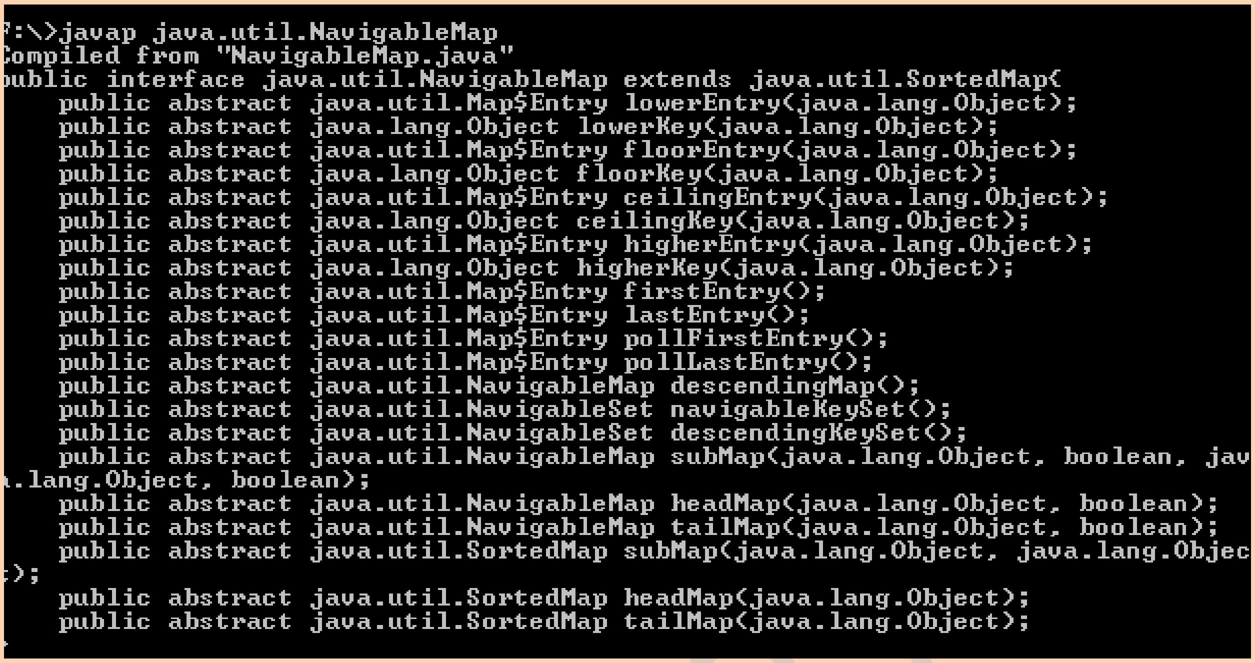
```
F:\>javap java.util.SortedMap
Compiled from "SortedMap.java"
public interface java.util.SortedMap extends java.util.Map{
    public abstract java.util.Comparator comparator();
    public abstract java.util.SortedMap subMap(java.lang.Object, java.lang.Objec
t);
    public abstract java.util.SortedMap headMap(java.lang.Object);
    public abstract java.util.SortedMap tailMap(java.lang.Object);
    public abstract java.lang.Object firstKey();
    public abstract java.lang.Object lastKey();
    public abstract java.util.Set keySet();
    public abstract java.util.Collection values();
    public abstract java.util.Set entrySet();
}
```

Here Comparator compartor() method will return null if sorting order is not an default sorting order.

NavigableMap:

It the child interface of SortedMap.

It having some methods like below:



```
F:\>javap java.util.NavigableMap
Compiled from "NavigableMap.java"
public interface java.util.NavigableMap extends java.util.SortedMap{
    public abstract java.util.Map$Entry lowerEntry(java.lang.Object);
    public abstract java.lang.Object lowerKey(java.lang.Object);
    public abstract java.util.Map$Entry floorEntry(java.lang.Object);
    public abstract java.lang.Object floorKey(java.lang.Object);
    public abstract java.util.Map$Entry ceilingEntry(java.lang.Object);
    public abstract java.lang.Object ceilingKey(java.lang.Object);
    public abstract java.util.Map$Entry higherEntry(java.lang.Object);
    public abstract java.lang.Object higherKey(java.lang.Object);
    public abstract java.util.Map$Entry firstEntry();
    public abstract java.util.Map$Entry lastEntry();
    public abstract java.util.Map$Entry pollFirstEntry();
    public abstract java.util.Map$Entry pollLastEntry();
    public abstract java.util.NavigableMap descendingMap();
    public abstract java.util.NavigableSet navigableKeySet();
    public abstract java.util.NavigableSet descendingKeySet();
    public abstract java.util.NavigableMap subMap(java.lang.Object, boolean, jav
a.lang.Object, boolean);
    public abstract java.util.NavigableMap headMap(java.lang.Object, boolean);
    public abstract java.util.NavigableMap tailMap(java.lang.Object, boolean);
    public abstract java.util.SortedMap subMap(java.lang.Object, java.lang.Objec
t);
    public abstract java.util.SortedMap headMap(java.lang.Object);
    public abstract java.util.SortedMap tailMap(java.lang.Object);
}
```

TreeMap:

- TreeMap if follows the one datastructe is called as RED-BLACK tree.
- Insertion order is not preserved but all the objects can be arranged according some sorting order of keys. Hence sorting order is preserved.
- Duplicate values cannot allow for keys, but values can be duplicate.
- If we use default sorting order, then the keys must be homogeneous and class must be implement Comparable interface, otherwise we will get a ClassCatException.
- If we use customized sorting order, then the keys need not be homogenous (heterogeneous) and class must be implements Comparator interface.

Null acceptance:

- For the empty TreeMap as the first element with null key is allowed, but after inserting null entry, we should not try to enter another null entries, if done we will get NullPointerException.

- For Non-empty TreeMap, we should not try enter any null entries, if done, we will get NullPointerException.
- There are no restrictions for values about null acceptance.
- If we not provide any sorting order then by default sorting order will be used.

<u>Constructors in TreeMap:</u>

(1) TreeMap t=new TreeMap();
(2) TreeMap t=new TreeMap(Comparator c);
(3) TreeMap t=new Treemap(Map m);
(4) TreeMap t=new TreeMap(SortedMap s);

<u>Properties:</u>

It is the chaild class of Hashtable class.

It can be used for representing key-values pairs.

Both key and values should be String objects.

Properties p =new Properties();

<u>Methods in Properties class:</u>

(1) String getProperty(String name):

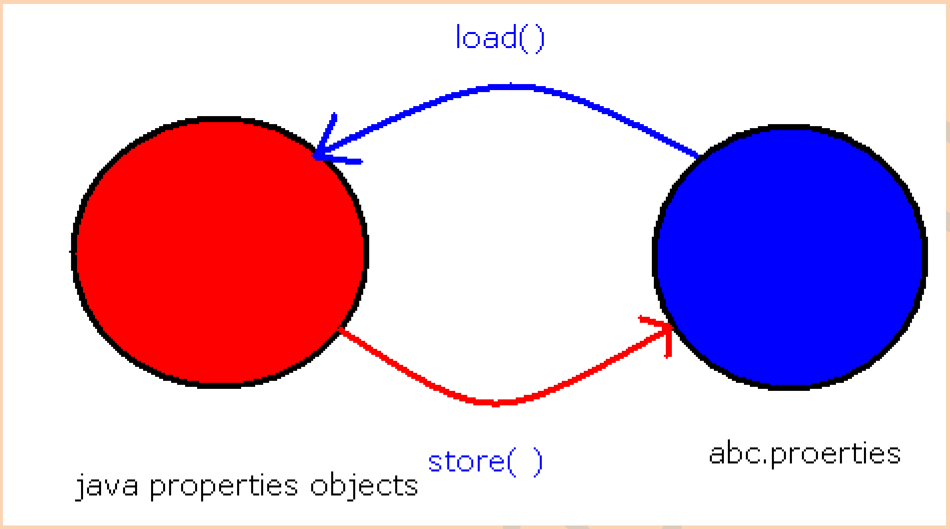
Returns the value associated with specified property.

If the specified property is not available, then it returns null values.

(2) String setProperty(String name, String value);
(3) Enumeration getPropertynames();
(4) Void load(InputStream is)

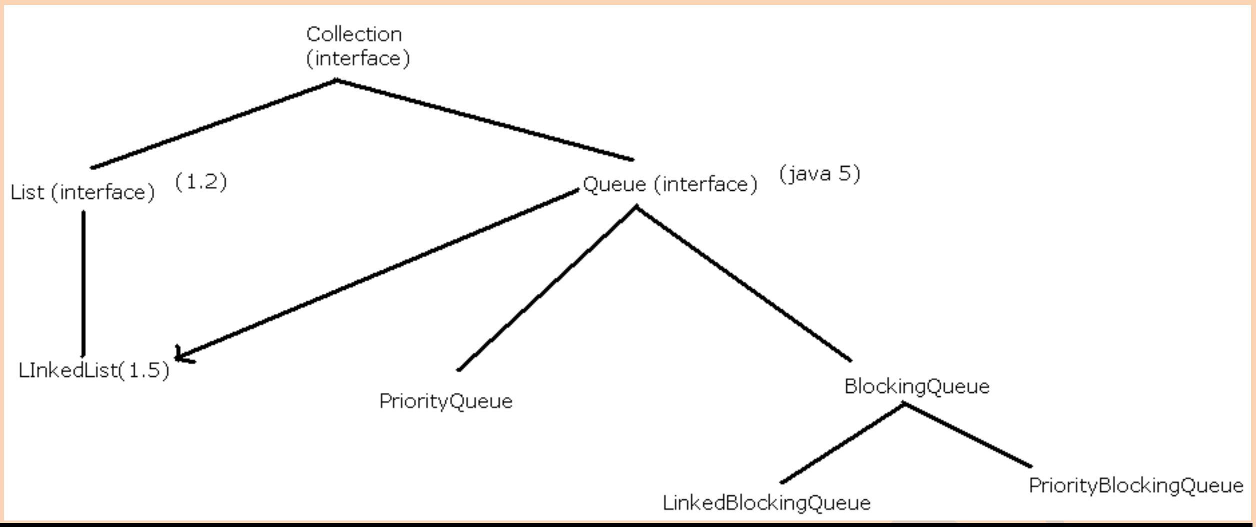
To load the properties from properties file to java properties object.

(5) Void store(OutputStream os, String comment);
To store the properties from java properties object to properties file.

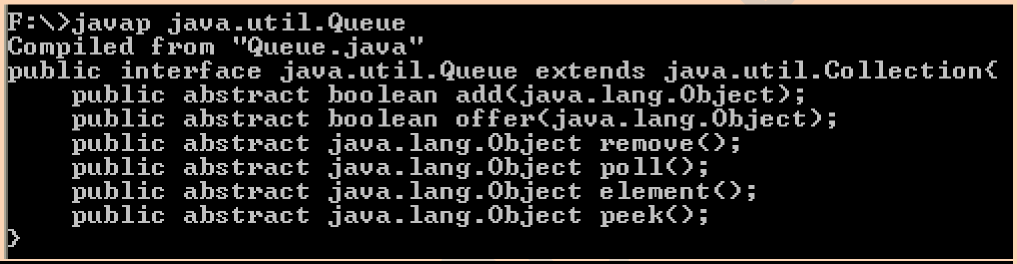


Queue Interface:

- If we want to represent a group of individual object prior to processing, then we should go for Queue interface.
- Queue in general follows FIFO order, but we can implement our own order by using PriorityQueues.
- LinkedList class is re-engineered in java5 version to implement Queue interface.
- LinkedList based implementation of queue always follows FIFO.



Methods in Queue:



```
F:\>javap java.util.Queue
Compiled from "Queue.java"
public interface java.util.Queue extends java.util.Collection{
    public abstract boolean add(java.lang.Object);
    public abstract boolean offer(java.lang.Object);
    public abstract java.lang.Object remove();
    public abstract java.lang.Object poll();
    public abstract java.lang.Object element();
    public abstract java.lang.Object peek();
}
```

PriorityQueue:

- If we want to store the objects prior to processing according to some priority, then we should go for PriorityQueue.
- Insertion order is not preserved. But all the elements are arranged according to some priority. It may be default natural storing order or customized sorting order specified by Comparator object.
- If your are depend upon the natural sorting, then elements should be homogeneous and comparable otherwise we will get ClassCastException.
- If we are depended upon the our own sorting order then elements need be homogeneous and comparable.
- Duplication objects are not allowed.

- Null insertion is not possible even the first element itself.

Constructors in PriorityQueue:

(1) PriorityQueue pq=new PriorityQueue();

Creates a default PriotyQueue, with the default size is 11 and all objects are sorted according some sorting order.

(2) PriorityQueue pq=new PriorityQueue(int initialcapacity , comparatory c);
(3) PriorityQueue pq=new PriorityQueue(SortingSet s);
(4) PriorityQueue pq=new PriorityQueue(Collection c);

Collections:

Collections is a utility class, provides several utility methods for the collection implemented classes.

(A) Sorting a List:

"Collections" class having a method to sorting a list i.e.,

Public static void sort (List L);

This method will sort the element of List by using some default sorting order.

If we use default sorting order, the elements must be homogeneous and comparable, otherwise we will get ClassCastException.

The List should not contain null values; otherwise we will get a NullPointerException.

Public void static sort (List l, comparator c)

This method will be useful for sorting the elements of List with some customized sorting order; here the elements need not homogeneous and comparable.

(2) Searching the List:

Public static int binarySearch(List l,Object key);

This method will be useful for searching the List elements with default searching order.

It is internally follows binarysearch algorithm.

Before using this method, the List elements should be sorted, otherwise we may get unreliable values.

The successful searching, this method will returns index value.

Otherwise it will show insertion place/index.

Insertion place/index is the place where we should enter some value to proper sorted.

Public static int binarySearch (List l, Object key, Comparator c);

This method will be useful for in the case of List is sorted by using Comparator.

We should pass the same Comparator object when we going to search the element from List.

Reversing the List elements:

Collection class contains one method for reversing the elements of List.

Public static void reverse (List l);

Arrays class:

Java.util package contain one utility class called Arrays.

This will provide some utility methods.

Sorting the elements of Array:

Following methods are helpful in sorting the Array elements.

Public static void sort(primitives[] p);

This method will sort the primitive elements of an array,

It will follow the default sorting order.

Public static void sort(Object[] o);

This method will sort the object type of elements of an Array..

It will maintain the default sorting order.

Public static void sort(Object[] o, comparator c);

To Sort the object Array according to customized sorting order.

Note:

The object Arrays will sort either customized sorting order or default sorting order, whereas primitive Arrays will sort only by natural/default sorting order only and not by customized sorting order.

Searching an Array:

(a) ***Public static int binarySearch(primitive[] p, primitive key);***
(b) ***Public static int binarySearch(Object[] o, Object key);***
(c) ***Public static int binarySearch(Object[] o, Object key, Comparator c);***

Converting of Arrays as a List:

- Arrays class contains one method to convert the Array in to List, i.e..
- Public static List asList(Objective[] o);
- This method won't create a List object, the Array object will be given in the form of List object.
- By using Array object, if we done any modification automatically it will reflect to List and same time if we done modification on List reference automatically it will reflect on Array object.
- By using List reference if we are performing any operation which varies the size we will get runtime error saying: UnsupporttedOoperationException.

Random Class:

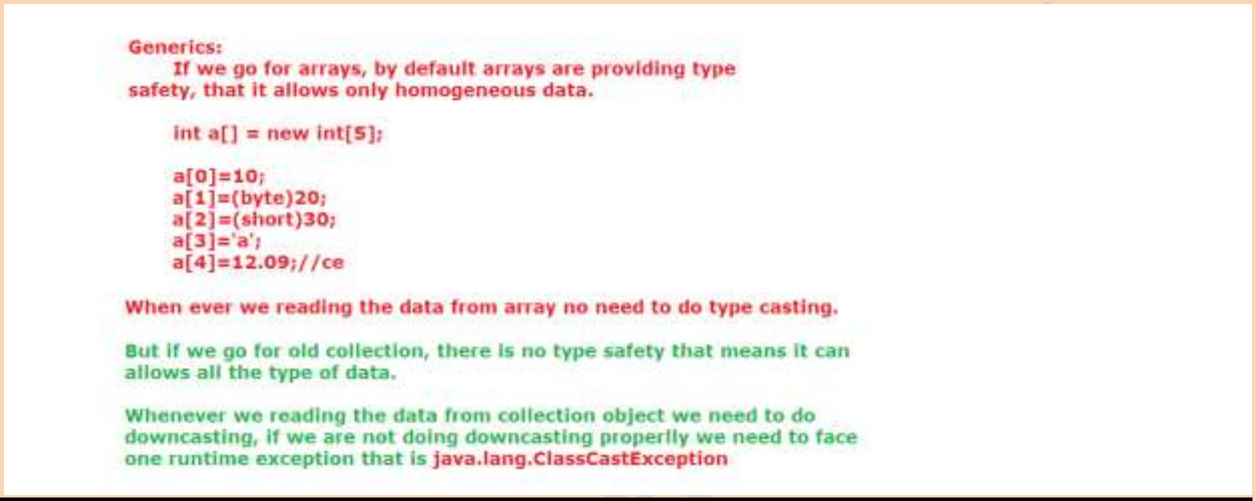
```
public class CollectionDemo {

        public static void main(String[] args){

        PriorityQueue pq = new PriorityQueue();

        Random r = new Random();

        int count = 0;

        while(true){

                int i = r.nextInt(1000);

                if(200<i && i<300){

                        System.out.println(i);

                        count++;

                        pq.offer(i);

                }
                if(count==10){
                        break;
                }
                        //System.out.println("    "+i);
        }
        }


}
```

Data and SimpleDateFormat:

```
import java.text.SimpleDateFormat;
import java.util.Date;
public class Demo{
        public static void main(String[] args){
                Date d = new Date();
                System.out.println(d);
                //SimpleDateFormat sdf = new SimpleDateFormat();
                //SimpleDateFormat sdf = new SimpleDateFormat("dd-
MM-yyyy");
                //SimpleDateFormat sdf = new SimpleDateFormat("dd-
MM-yy");
                //SimpleDateFormat sdf = new SimpleDateFormat("dd-
MMMM-yy");

                //SimpleDateFormat sdf = new SimpleDateFormat("dd-
MMM-yy");
                //SimpleDateFormat sdf = new
SimpleDateFormat("dddd-MMM-yy");
                SimpleDateFormat sdf = new
                                SimpleDateFormat("dd-MMM-yy
hh:mm:ss");
                String s = sdf.format(d);
```

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

Generic Programs:



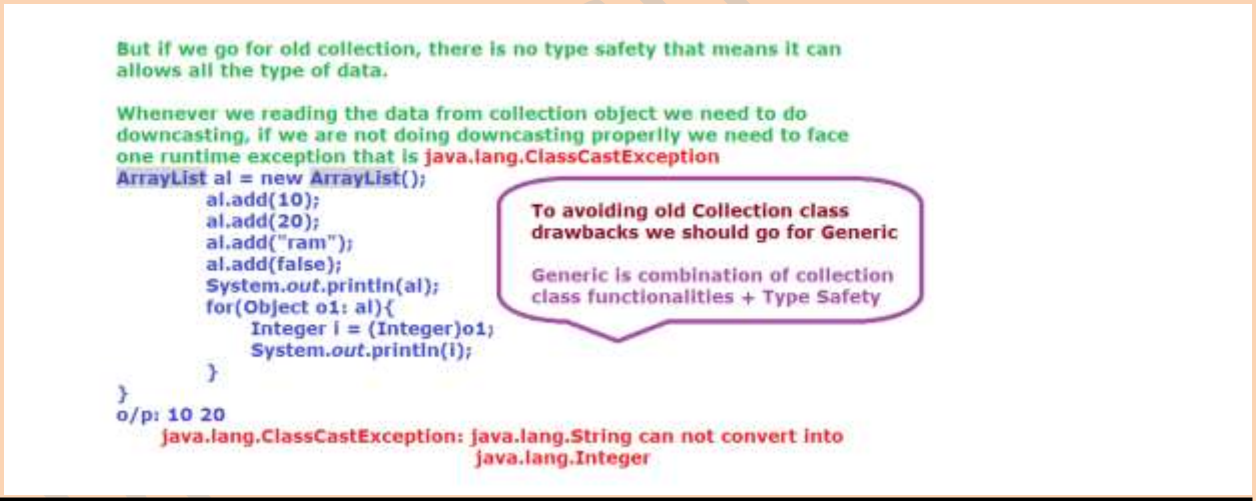
Generics:
If we go for arrays, by default arrays are providing type safety, that it allows only homogeneous data.

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

a[0]=10;
a[1]=(byte)20;
a[2]=(short)30;
a[3]='a';
a[4]=12.09;//ce
```

When ever we reading the data from array no need to do type casting.

But if we go for old collection, there is no type safety that means it can allows all the type of data.

Whenever we reading the data from collection object we need to do downcasting, if we are not doing downcasting properly we need to face one runtime exception that is java.lang.ClassCastException



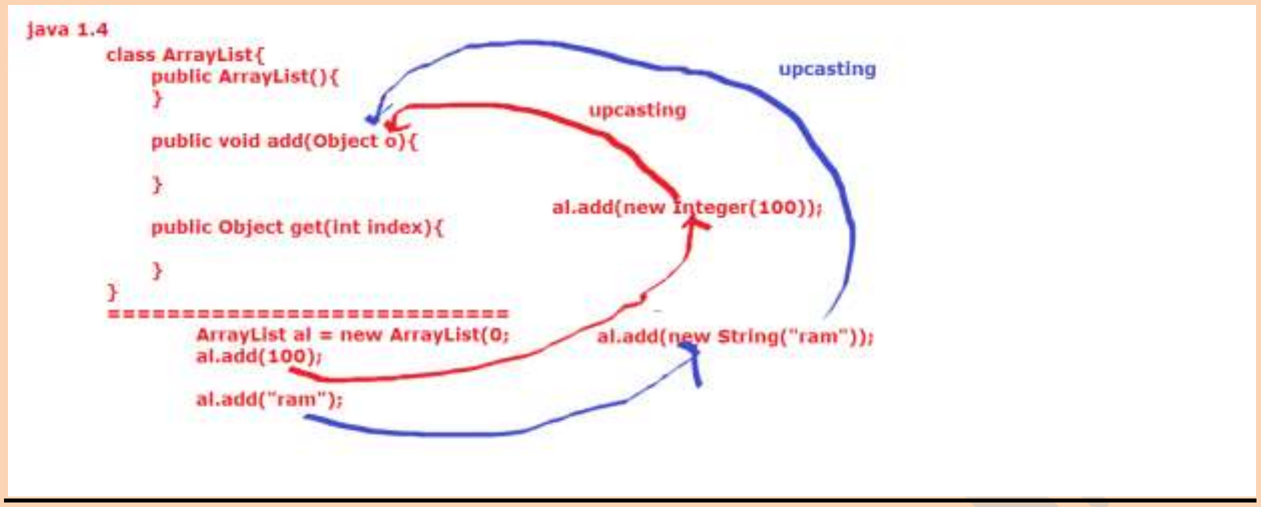
But if we go for old collection, there is no type safety that means it can allows all the type of data.

Whenever we reading the data from collection object we need to do downcasting, if we are not doing downcasting properly we need to face one runtime exception that is java.lang.ClassCastException

```
ArrayList al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add("ram");
        al.add(false);
        System.out.println(al);
        for(Object o1: al){
            Integer i = (Integer)o1;
            System.out.println(i);
        }
}
o/p: 10 20
    java.lang.ClassCastException: java.lang.String can not convert into
                                  java.lang.Integer
```

To avoiding old Collection class drawbacks we should go for Generic

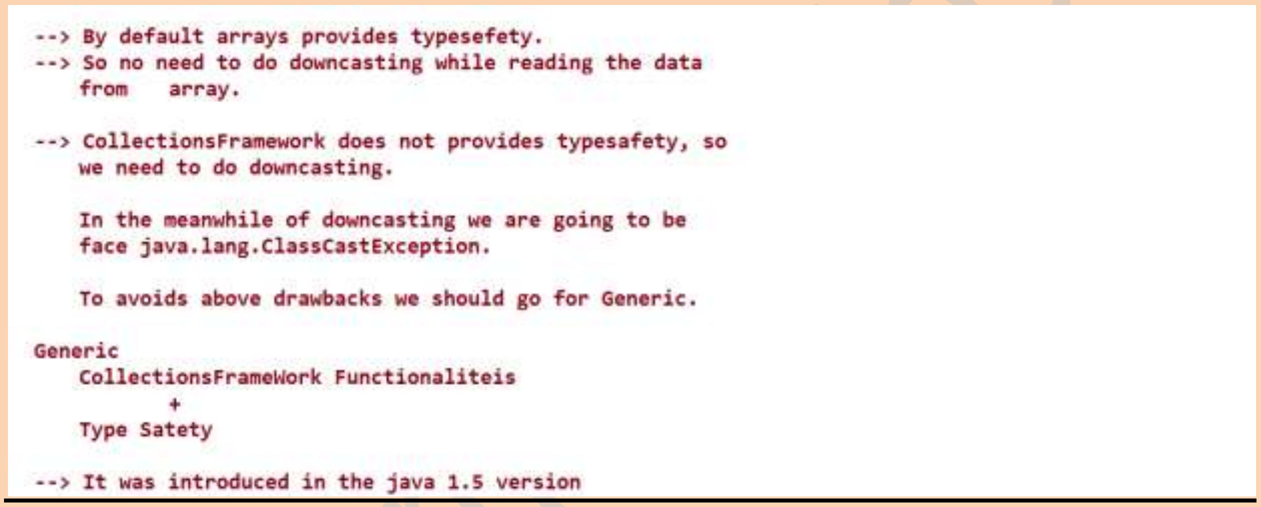
Generic is combination of collection class functionalities + Type Safety



```
java 1.4
        class ArrayList{
            public ArrayList(){
            }

            public void add(Object o){

            }

            public Object get(int index){

            }
        }
        ============================
                ArrayList al = new ArrayList(0;
                al.add(100);

                al.add("ram");
```

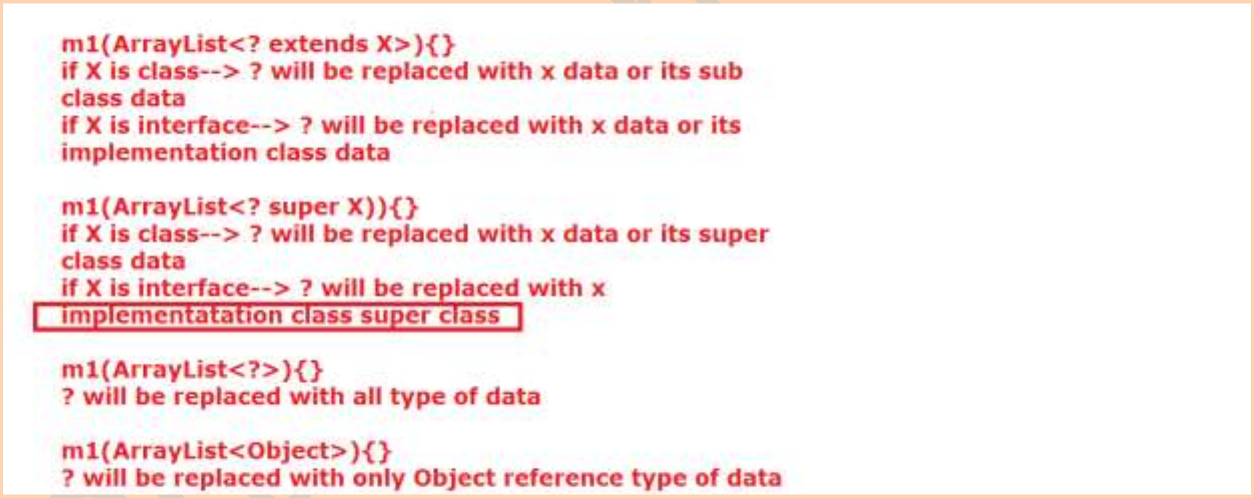
```
al.add(new Integer(100));
```

```
al.add(new String("ram"));
```



```
--> By default arrays provides typesefety.
--> So no need to do downcasting while reading the data
    from   array.

--> CollectionsFramework does not provides typesafety, so
    we need to do downcasting.

    In the meanwhile of downcasting we are going to be
    face java.lang.ClassCastException.

    To avoids above drawbacks we should go for Generic.

Generic
    CollectionsFrameWork Functionaliteis
            +
    Type Satety

--> It was introduced in the java 1.5 version
```

```
package generic;

class Test<T>{

        T i;

        Test(T i){

                this.i = i;

        }

        T getValue(){

                return i;
        }
}
public class GenericDemo {
        public static void main(String[] args) {
                Test<Integer> t = new Test<Integer>(10);
                System.out.println(t.getValue());
                Test<String> t1 = new Test<String>("Nikhil");
                System.out.println(t1.getValue());
        }
}
```



m1(ArrayList<? extends X>){}
if X is class--> ? will be replaced with x data or its sub class data
if X is interface--> ? will be replaced with x data or its implementation class data

m1(ArrayList<? super X)){}
if X is class--> ? will be replaced with x data or its super class data
if X is interface--> ? will be replaced with x implementatation class super class

m1(ArrayList<?>){}
? will be replaced with all type of data

m1(ArrayList<Object>){}
? will be replaced with only Object reference type of data

```
package generic;
class Test<T extends Number>{
        T i;
        Test(T i){
                this.i = i;
        }
        T getValue(){
                return i;
        }
}
public class GenericDemo {
        public static void main(String[] args) {
                Test<Integer> t = new Test<Integer>(10);
                System.out.println(t.getValue());
                /*Test<String> t1 = new Test<String>("Nikhil");
                System.out.println(t1.getValue());*/
                Test<Float> t1 = new Test<Float>(20.0f);
                System.out.println(t1.getValue());
        }
}
=====================
package generic;
class Test<T extends Runnable>{
        T i;
        Test(T i){
                this.i = i;
        }
        T getValue(){
                return i;
        }
        void execute(){
                Thread t = new Thread(i);
                t.start();
        }
}
class MyThread implements Runnable{
        public void run(){
                System.out.println("run method");
        }
}
public class GenericDemo {
        public static void main(String[] args) {
                Test<Thread > t = new Test<Thread>(new Thread());
                System.out.println(t.getValue());
                MyThread mt = new MyThread();
                Test<MyThread> t1 = new Test<MyThread>(mt);
                System.out.println(t1.getValue());
                t1.execute();
        }
```

```
}
========================
package generic;
import java.util.ArrayList;
class A{
	int a =111;
}
class B extends A{
	int b = 222;
}
public class GenericDemo {
	static void m1(ArrayList<? extends Number> al){
		System.out.println("m1 method");
		System.out.println(al);
	}
	public static void main(String[] args) {
		ArrayList<Integer> al = new ArrayList<Integer>();
		al.add(100);
		al.add(200);
		//al.add("ram");
		m1(al);
		ArrayList<Double> al2 = new ArrayList<Double>();
```

```
        al2.add(100.00d);
        al2.add(200.00d);
        m1(al2);
        ArrayList<String> al1 = new ArrayList<String>();
        al1.add("ram");
        al1.add("kiran");
        //al1.add(false);
        //m1(al1);
    }
}
=============================
package generic;
import java.util.ArrayList;
class A{
    int a;
    A(int a){
        this.a = a;
    }
    A(){}
    public String toString(){
        return a+" ";
    }
```

```
}
class B extends A{
        String b;
        B(int a , String b){
                super();
                this.a = a;
                this.b = b;
        }
        public String toString(){
                return a+"...."+b;
        }
}
public class GenericDemo {
        static void m1(ArrayList<?> al){
                System.out.println("m1 method");
                System.out.println(al);
        }
        public static void main(String[] args) {
                ArrayList<Integer> al1 = new ArrayList<Integer>();
                al1.add(10);
                al1.add(20);
                al1.add(30);
                m1(al1);

                ArrayList<String> al2 = new ArrayList<String>();

                al2.add("ram");

                al2.add("suji");

                m1(al2);

        }

}
```

enum:

enum is to used grouping the named constants.

With the help of enum we can develop user define datatypes.

It has been introduced in java 1.5 version.

Ex:

```
enum Color{
        BLUE,GREEN,YELLOW;
}
```

enum is always followed by enum_name.

We can write empty enums also.

Ex:

```
enum Color{
}
```

Naming conventions of enum same as class.

Ex:

```
enum Month{
        jan,feb,march;
}
```

We can write the enum constants either in small letters or in capital letters or in combinations.

Constants may ended with the semicolon(;). It not a mandatory.

Ex:

```
enum Tifin{
        Idly,Dosa


}
```

If we want write any other content otherthan constants the constants must be ended with semicolon.

Ex:

```
enum Tifin{
 Idly,Dosa; //here ';' is mandatory.
 void m1(){
 }
}
```

We cannot write abstract methods in constants.

```
enum Tifin{
        Idly;
        //abstract void m1();
}
```

If we want to write any other content without constants in enum, before the content we should be place semicolon(;).

Ex:

```
enum Tifin{
        ; //mandatory
        void m1(){
        }
}
```

Within the enum we can write main method, we can compile and execute the programs.

Without class definition we can write the java code we can compile and execute.For this purpose we have two ways.

1. with enum
2. with interface

1. with enum (from java 1.5 onwards)

```
        enum Color{
                ;
        public static void main(String args[]){
          System.out.println("we can");
        }
}
```

2. with interface (from java 1.8)

```
interface I{
        public static void main(String args[]){
                System.out.println("we can");
         }
}
```

We can write the enums outside of the class and inside of the class.

If we are writing enum outside of the class, the bellow modifiers are allowed.

Ex: public, package-private (default), strictfp

If we are writing enum inside of the class, the bellow modifersare allowed.

Ex: public,package-private(default), strictfp, private, protected, static,

but we can not write final.

In above two conditions we can not write abstract keyword.

Every user define enum is by default subclass of java.lang.Enum.

We cannot write extends keyword after enum name, but we can write implements keyword.

Ex:

```
enum Color extends Object{//wrong syntax

}
```

Ex:

```
enum Color implements Runnable{

        ;
        public void run(){
        }
    }
```

From java 1.5 onwards we can use enum constants value in the switch case also.

"case" variables names must be equal or less than enum constants, and spelling must be equal (including lower and upper case).

We can not create object for enum.

But we can create reference for enum.

Ex:

```
    enum Color{
        ;
        public static void main(String[] ram){
        //Color c = new Color; //wrong syntax
        Color c;
        }
    }
```

With the help of enum reference we can hold constant values.

If we want read values from enum, we have one method like values().

If we want know the positions of constants, then we have one method like ordinal().

```
Ex: class A{
        enum Color{
                Blue,Yellow,Red;
        }
        public static void main(String... ram){
                Color c[] = Color.values();
                for(Color c1 : c){
                        S.o.p(c1+"..."+c1.ordinal());
                }
        }
}
```

```
package enums;
enum Color{
        BLUE,GREEN,YELLOW,red
}
enum Flower{
        ;
        void m3(){
        }
}
/*enum A extends java.lang.Object{
} */
```

```
enum B implements java.lang.Runnable{
        ;
        public void run(){
        }
}
public class EnumDemo {
        enum Month{
                Jan,Feb;
                void m1(){
                }
                //abstract void m2();
        }
        public static void main(String[] args) {
                //Color c = new Color;
                Color c[]= Color.values();
                for(Color c1: c){
                        System.out.println(c1+"..."+c1.ordinal());
                }
        }
}
```

enum constants by default public static final.

enum with switch case:

```
enum Color{
	BLUE,GREEN,YELLOW,RED;
}
public class EnumDemo {
	public static void main(String[] args) {
		Color c = Color.YELLOW;
		switch(c){
		case BLUE: System.out.println("THIS IS BLUE 
COLOR");
				break;
		case GREEN: System.out.println("THIS IS GREEN 
COLOR");
			break;
		case YELLOW: System.out.println("THIS IS  YELLOW 
COLOR");
			break;
	/*case RED: System.out.println("THIS IS RED COLOR");
			break;*/
	/*case PINK: System.out.println("THIS IS PINK COLOR");
				break;*/
	default : System.out.println("NO COLOR MATCHED");
}}}
```

/*switch statement case values must same as enum constants. We cannot use other than enum constant values, but we can decrease the enum constants.*/

```
import java.util.Scanner;



public class EnumDemo {
	enum Month{
		jan(31),feb,march(31);
		int noofdays;
		Month(){
			Scanner scan = new Scanner(System.in);
			System.out.println("enter days");
			int x = scan.nextInt();
			//noofdays=28;
			noofdays = x;
		}
		Month(int days){
			noofdays = days;
		}
		int getDays(){
			return noofdays;
		}
	}
	public static void main(String[] r){
		Month[] m = Month.values();
		for(Month m1: m){
			System.out.println(m1+"..."+m1.getDays());
		}
	}
}
```