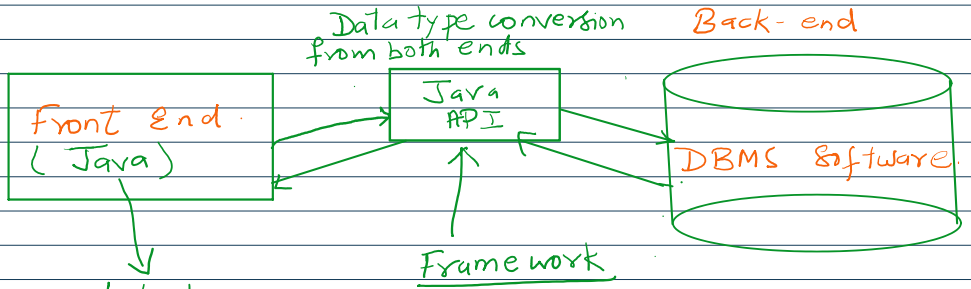
Collection (API)

20 January 2025 18:23

Application Program Interface(API) it is a framework of classes and interfaces. It is used to solve complex problems.



⇒ JDBC ← 3rd party ← download.

MongoDB / Datatype

Collection swing — Framework 1st party

Framework 1st party

Javafx ← 3rd party

↑ Cross-platform.

Stride ← 3rd party

Array

An array is an indexed collection of fixed number Homogenous data elements.
The main advantage of arrays is, we can represent multiple values, by using single variable. So that readability of code will be improved.

Limitations of Arrays

1. **Arrays are fixed in size. That is, once we create an array, there is no chance of increasing or decreasing the size based on our requirement. due to this to use array concept compulsorily, we should know the size in advance, which may not be possible always.**
2. **Arrays can hold homogeneous data type elements.**

```
Student[] s = new Student[10000];

s[0] = new Student();
for(I = 0; I < 10000; I++){
   s[I] = new Student();
}
```

How to store heterogenous elements in an Array?

```
Object[] A = new Object[10000];

A[0] = new Student();
A[1] = new Customer();
A[2] = new Employee();
```

Array concept is not implemented based on some standard data structure. And hence, ready-made method support is not available for every requirement, we have to write the code explicitly, which increases the complexity of programming.

To overcome above problems of arrays, we should go for Collection concepts.

- **Collections are growable or shrinkable in nature, that is based on our requirement. We can increase or decrease the size.**
- **Collection can hold both homogeneous and heterogeneous objects.**
- **Every collection class is implemented based on some standard data structure. Hence, for every requirement, ready-made method support is available.**
- **Being a programmer, we are responsible to use those methods, and we are not responsible to implement those methods.**

Difference between Arrays and Collection

Arrays	Collection
Fixed in size. That is, once we create an array, we can't increase or decrease the size based on our requirements.	Growable in nature, that is based on our requirement. We can increase or decrease the size
With respect to memory arrays are not recommended to use.	With respect to memory collection are recommended to use.
With respect to performance Arrays recommended to use.	With respect to performance collection are not recommended to use.
Arrays can hold only homogeneous data type elements.	Collection can hold both homogeneous and heterogeneous elements.
No underlying data structure for arrays. And hence, ready-made method support is not available. for every requirement, we have to write the code explicitly which increases complexity of programming.	Every collection class is implemented based on some standard data structure. And hence, for every requirement, ready-made methods support is available. Being a programmer, we can use these methods directly, and we are not responsible to implement those methods.
Arrays can hold both primitives and object types.	Collection can hold only object types, but not primitives.

What is Collection?
If we want to represent a group of individual objects as a single entity, then we should go for collection.

Collection Framework: It contains several classes and interfaces which can be used to represent a group of individual objects as a single entity.

Java Collection ≠ C++ Container

Collection Framework → STL (Standard Template Library)

9 Key interfaces of Collection Framework.

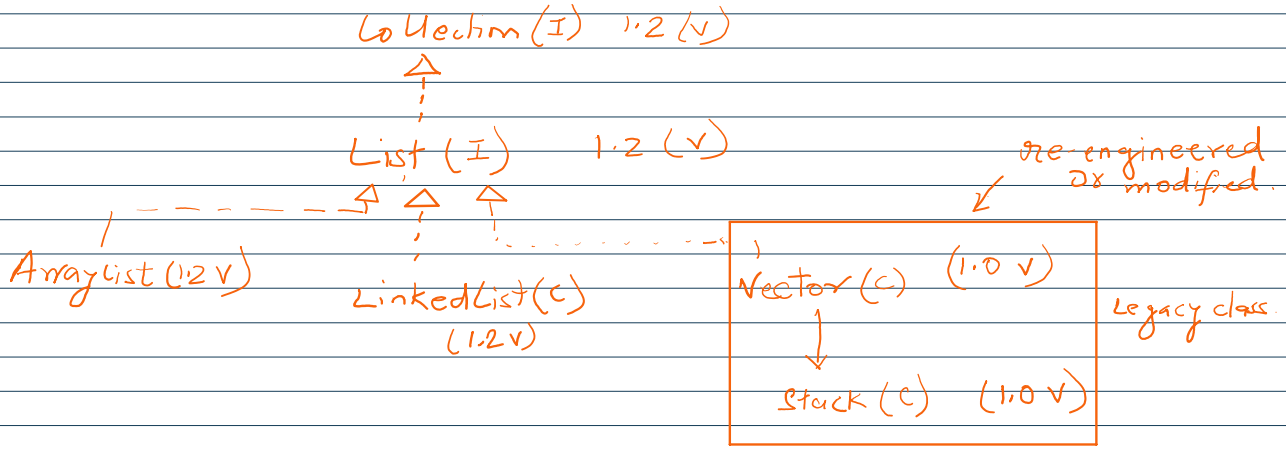
1. Collections 2. List 3. Set 4. SortedSet
5. NavigableSet 6. Queue 7. Map 8. SortedMap
9. NavigableMap

Collection(I): defines the most common methods which are applicable for any collection object. In general Collection(I) is considered as root(I) of Collection framework. There is no concrete class, which implements Collections(I) directly.

What is the difference between Collection and Collections?
Collection is an interface. If we want to represent a group of individual object in a single

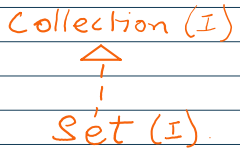
entity, then we should go for Collection interface.

Collections(C) is an utility class present in java.util package to define several utility methods for Collections objects(like sorting, search, etc.)

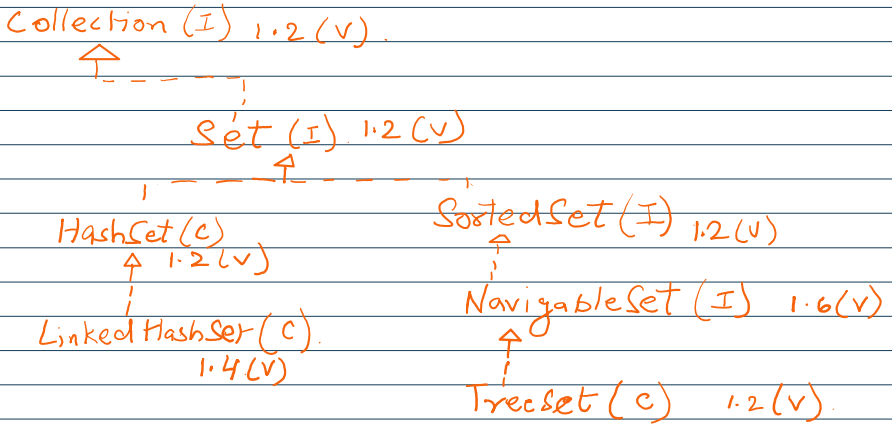


List(I): It is a child interface of Collection(I). If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order must be preserved, then we should go for list interface.

Note: In 1.2 version Vector & Stack classes are Re-engineered or modified to implement List interface.



SET(I): It is a child interface of Collection. If we want to represent a group of individual Object as a single entity where it duplicates are not allowed and insertion order not required to preserve, then we should go for set interface.

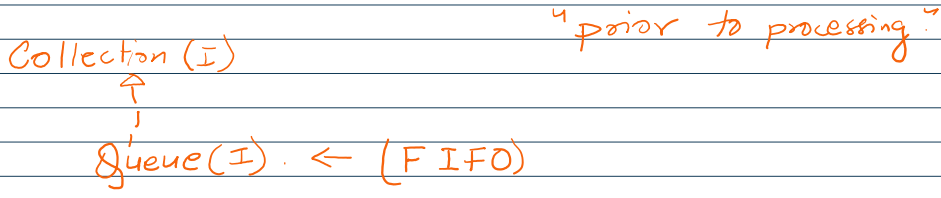


Sorted Set(I): It is the child interface of **Set** interface. If we want to represent a group of individual object as a single entity, where it duplicates are not allowed, and all objects should be inserted according to some sorted order, then we should go for **SortedSet** interface.

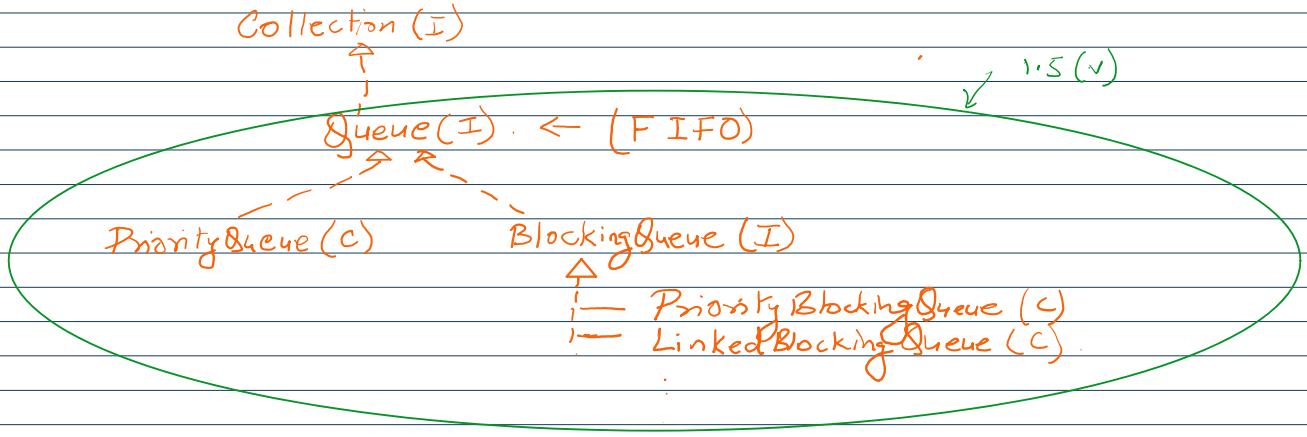
NavigableSet(I): It is a child interface of **SortedSet** interface. It contains several methods for navigation purposes (Scrolling of data).

What is the difference between Set(I) and List(I)

Set(I)	List(I)
Duplicates are not allowed.	Duplicates are allowed.
Insertion order is not preserved.	Insertion order is preserved.



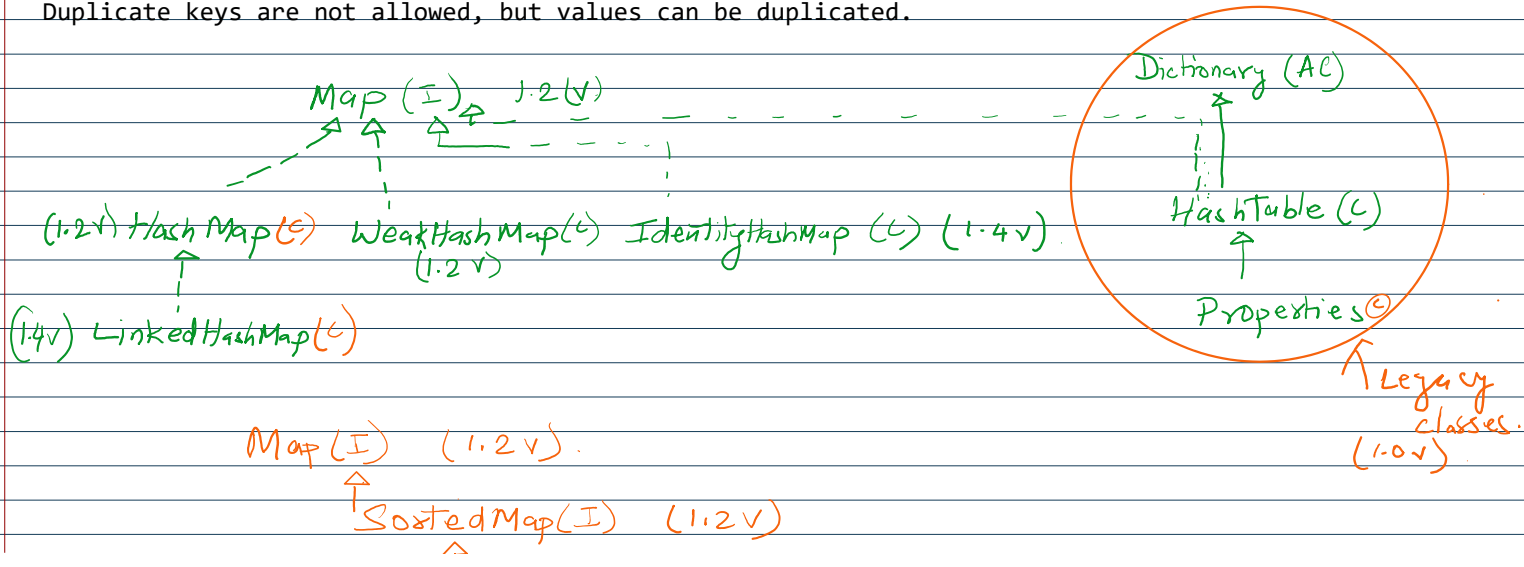
Queue(I): Queue is a child interface of collection interface. If we want to represent a group of individual "prior to processing" then we should go for Queue. Usually queue follows FIFI order. But based on our requirement, we can implement our own priority order also. Example, before sending a mail, all mail-ids have to store in some data structure in which order we added mail-id, the same ordering mail should be delivered for this requirement. Queue is the best choice.

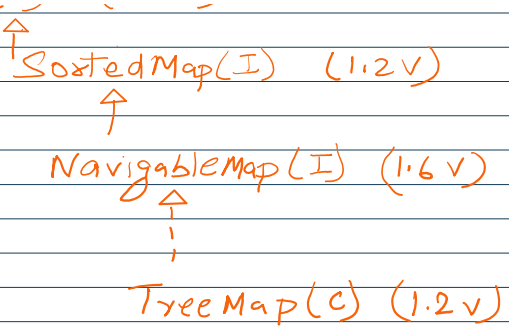


Note: All the above interfaces (Collection, List, Set, SortedSet, NavigableSet. Queue) ment for representing a group of individual objects.

If we want to represent a group of objects as key-value pair, then we should go for Map interface.

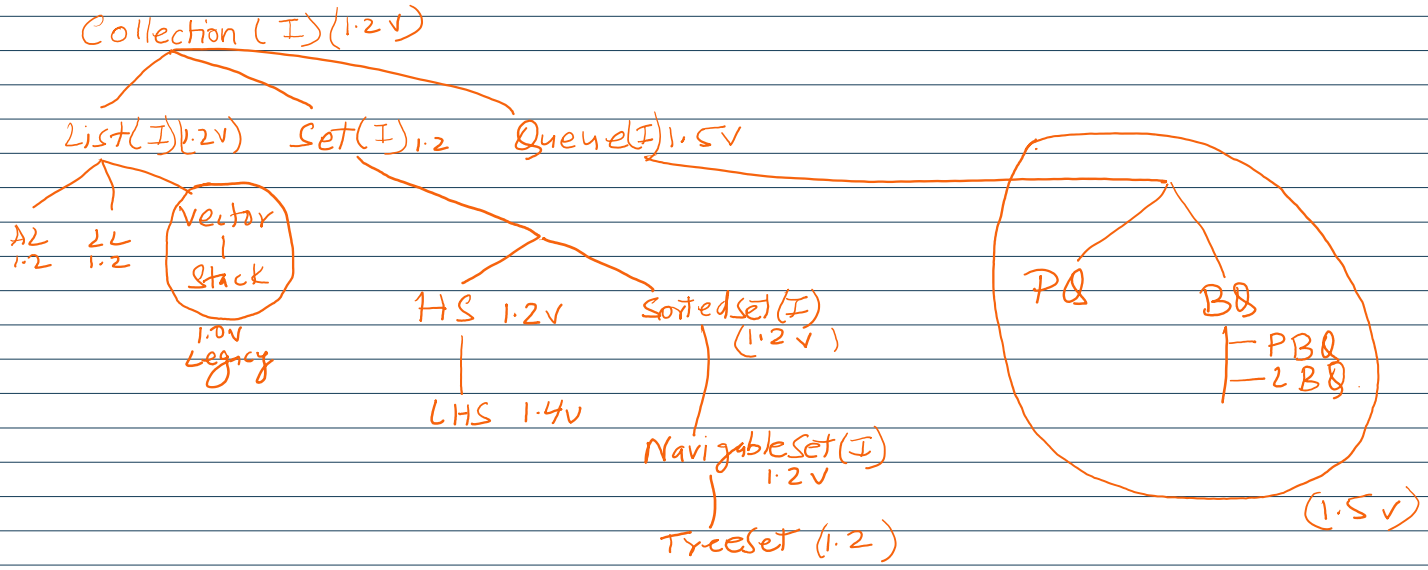
Map(I): Map is not child interface of collection. If we want to represent a group of object as key value pairs, then we should go for map interface. both key and value are objects only. Duplicate keys are not allowed, but values can be duplicated.

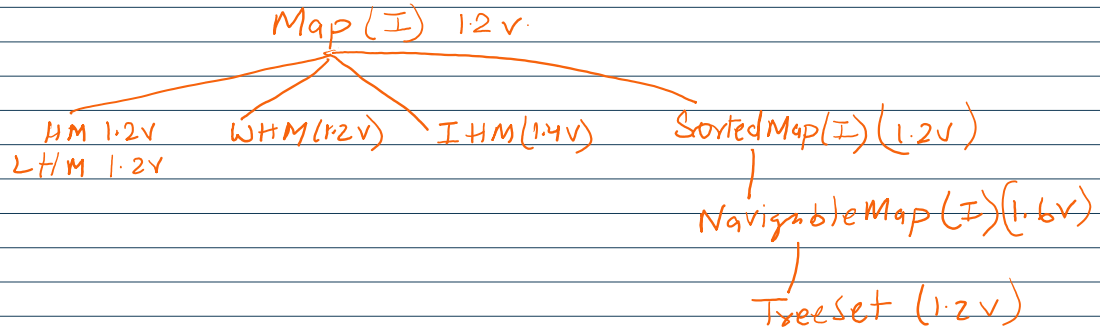




SortedMap(I): It is a child interface of **Map** interface. If we want to represent a group of key-value pairs according to some sorting order of keys, then we should go for **SortedMap** interface. In sorted map interface, the shorting should be based on key, but not based on value.

NavigableMap(I): It is a child interface of **SortedMap** interface. It defines several methods for navigation purpose.





The following are legacy characters present in collection framework:
Enumeration interface.
Dictionary abstract class.
Vector class.
Stack class.
Hashtable class.
Properties class.

Sorting:
Comparable(I): Default natural sorting order(Ascending order).
Comparator(I): Customized sorting order.

Cursors:
Enumeration(I)
Iterator(I)
ListIteraor(I)

Utility Classes:
Collections
Arrays

Collection interface

28 January 2025 20:21

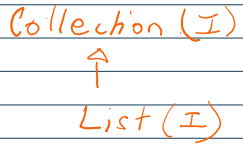
If we want to represent a group of individual objects as a single entity, then we should go for Collection.
Collection interface defines the most common methods which are applicable for any Collection object.

```
boolean add(Object o)
boolean addAll(Collection c)
boolean remove(Object o)
boolean removeAll(Collection c)
boolean retainAll(Collection c) => to remove all objects except those present in c.
void clear()
boolean contains(Object o)
boolean containsAll(Collection c)
boolean isEmpty()
int size()
Object[] toArray()
Iterator iterator()
```

Note: There is no concrete class which implements Collection interface directly.

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Collection Link for more study.

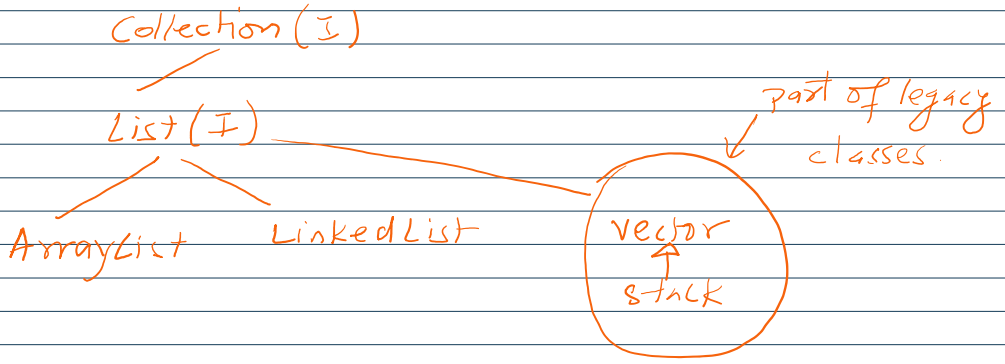


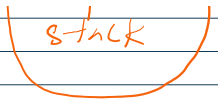
List interface:

List is a child interface of Collection interface. If we want to represent a group of individual object as a single entity, where duplicates are allowed and insertion order must be preserved, then we should go for List interface.

We can preserve insertion order via or with index. And we can differentiate, duplicate object by using index. Hence, index will play very important role in the list.
List interface defines the following specific methods:

```
void add(int index, Object o)
boolean addAll(int index, Collection c)
Object get(int index)
Object remove(int index)
Object set(int index, Object newObj) => To replace the element present at the specified index with provided object and returns old object.

int indexOf(Object o) => returns index of first occurrence of 'o' and -1 if 'o' is not present.
int lastIndexOf(Object o)

ListIterator listIterator()
```





ArrayList is a class:
1. Resizable Array or Growable or shrinkable Array
2. Duplicates are allowed.
3. Insertion order is preserved.
4. Can hold Heterogenous objects.(except TreeSet and TreeMap objects)
5. "null" insertion is allowed.

Constructors:

ArrayList al = new ArrayList(); => Creates an empty array list object with default initial capacity 10. Once array list reaches its maximum capacity, then a new array list object will be created with new capacity.

newCapacity = currentCapacity * 3/2 + 1

ArrayList al = new ArrayList(int initalCapacity); => Creates an empty array list object with a specified initial capacity.

ArrayList al = new ArrayList(Collection c); => Creates an equivalent array list object for the given collection.

```
package ArraysProgram;
import java.util.ArrayList;
public class ArrayListOne {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("A");
        al.add(10);
        al.add("Raghav");
        al.add(Math.PI);
        System.out.println(al);
        al.remove(1);
        System.out.println(al);
        al.add(2, Integer.MAX_VALUE);
        al.add("Shiv");
        System.out.println(al);
        System.out.println("Last element: "+al.get(al.size()-1));
    }
}
```

Usually we can use collection to hold and transfer object from one location to another location (Container). To provide support for this requirement every collection class by default implements Serializable and Cloneable interface.

ArrayList & Vector are the class that implements RandomAccess Interface.

RandomAccess interface: present in java.util package and it doesn't contain any method. It is a marker interface where required ability automatically used by the JVM. The primary purpose of this interface is to allow generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access.
This interface is a member of the Java Collections Framework.

Since:
1.4

```java
package Chapter1;


/**
 * Write a description of class Che here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
import java.io.Serializable;
import java.util.*;

public class CheckingInterfaces {
    public static void main(String[] args) {
        ArrayList l1 = new ArrayList();
        LinkedList l2 = new LinkedList();
        Vector v = new Vector();
        System.out.print("\f");
        System.out.println(l1 instanceof Serializable);
        System.out.println(l2 instanceof Serializable);
        System.out.println(l1 instanceof Cloneable);
        System.out.println(l2 instanceof Cloneable);
        System.out.println(l1 instanceof RandomAccess);
        System.out.println(l2 instanceof RandomAccess); //false
        System.out.print("\n" + (v instanceof Serializable));
        System.out.print("\n" + (v instanceof Cloneable));
        System.out.print("\n" + (v instanceof RandomAccess));
    }
}
```

BlueJ: Terminal Window - Col

Options

```
true
true
true
true
true
false
true
true
true
```

Class compiled - no syntax errors

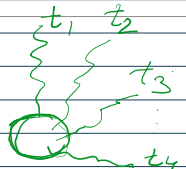
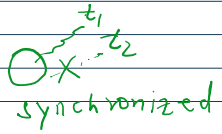
List has implemented
Linked ⇒ 1. Serializable
⇒ 2. Cloneable
it hasn't implemented RandomAccess

ArrayList & vector has 1. Serializable
2. Cloneable
3. RandomAccess
implement all 3 interfaces

ArrayList is the best choice if our frequent operation is retrieval operation. Because ArrayList implements random access interface.
ArrayList Is the worst choice if our frequent operation is insertion or deletion in the middle.

What is the difference between ArrayList and Vector?

Threads.

ArrayList	**Vector**
Every method presented in the array list is non-synchronized.	Every method present in the vector is synchronized.
 (better performance) non synchronized	At a time, only one thread is allowed to operate on vector object, and hence it is thread safe.  synchronized
Add a time multiple threads are allowed on arrow list object, and hence it is not thread safe.	
A relatively performance is high because threats are not required to wait for the operation on array list object.	A relatively performance is low because threats are required to wait to operate on vector object.
Introduced in version 1.2. And it is not a legacy class.	Introduced in 1.0 version. And it is a legacy class.

How to get synchronized version of ArrayList object?

List (I)
⇧
ArrayList

```
ArrayList al = new ArrayList();
List l = Collections.synchronizedList(al);
```

l ↑ Synchronized

By default, ArrayList is non synchronized, but we can get synchronized version of array list object by using synchronizedList() method of Collections class.

public static <T> List<T> synchronizedList(List<T> list)
Returns a synchronized (thread-safe) list backed by the specified list. In order to guarantee serial access, it is critical that **all** access to the backing list is accomplished through the returned list.
It is imperative that the user manually synchronize on the returned list when iterating over it:

```
  List list = Collections.synchronizedList(new ArrayList());
      ...
  synchronized (list) {
      Iterator i = list.iterator(); // Must be in synchronized block
      while (i.hasNext())
          foo(i.next());
  }
```

Failure to follow this advice may result in non-deterministic behavior.
The returned list will be serializable if the specified list is serializable.

Type Parameters:
T - the class of the objects in the list
Parameters:
list - the list to be "wrapped" in a synchronized list.
Returns:
a synchronized view of the specified list.

From <https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList-java.util.List->

Vector:
1. Underline data structure is resizable array or growable array.
2. Insertion order is preserved.
3. Duplicates are allowed.
4. Heterogeneous objects are allowed.
5. Null insertion is possible.
6. It implements Serializable, Cloneable and RandomAccess interfaces.
7. Every method present in the vector is synchronized, and hence vector object is thread safe.

Constructors:
Vector v = new Vector(); => Creates an empty vector object with default initial capacity of 10. Once vector reaches its max capacity, then a new vector object will be created with. new capacity: (new_capacity = current_capcity * 2) Double the capacity.

Vector v = new Vector(int initialCapacity); => Creates an empty vector object with the specified initial capacity.

Vector v = new Vector(int initialCapacity, int incrementalCapacity);

Vector v = new Vector(Collection c); => Creates an equivalent object for the given collection. This constructor meant for interconversion between collection objects.

Vector Specific Methods

To add Objects:

boolean	**add**(E e) Appends the specified element to the end of this Vector.
void	**add**(int index, E element) Inserts the specified element at the specified position in this Vector.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified Collection into this Vector at the specified position.
void	**addElement**(E obj) Adds the specified component to the end of this vector, increasing its size by one.

To remove objects:

void	**clear()** Removes all of the elements from this Vector.
E	**remove(int index)** Removes the element at the specified position in this Vector.
boolean	**remove(Object o)** Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
boolean	**removeAll(Collection<?> c)** Removes from this Vector all of its elements that are contained in the specified Collection.
void	**removeAllElements()** Removes all components from this vector and sets its size to zero.
boolean	**removeElement(Object obj)** Removes the first (lowest-indexed) occurrence of the argument from this vector.
void	**removeElementAt(int index)** Deletes the component at the specified index.
boolean	**removeIf(Predicate<? super E> filter)** Removes all of the elements of this collection that satisfy the given predicate.
protected void	**removeRange(int fromIndex, int toIndex)** Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.

To get Objects:

E	**get(int index)** Returns the element at the specified position in this Vector.
E	**elementAt(int index)** Returns the component at the specified index.
E	**firstElement()** Returns the first component (the item at index 0) of this vector.
E	**lastElement()** Returns the last component of the vector.

Other methods:

```
int size();
int capacity();
Enumeration elements();
```

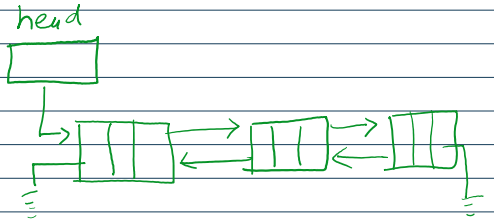
```java
package Chapter1;
import java.util.Vector;


/**
 * Write a description of class VectorDemo1 here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class VectorDemo1
{
    public static void main(String args[]){
        Vector v = new Vector();
        System.out.print("\f"+v.capacity()); //10
        for(int i = 1; i <= 10; i++){
            v.addElement(i);
        }
        System.out.print("\n"+v.capacity()); //10
        v.addElement("A");
        System.out.print("\n"+v.capacity()); //20
        System.out.print("\n" + v);
    }
}
```

```
BlueJ: Terminal Window - CollectionFramework
Options
10
10
20
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
```

Linked List

10 February 2025 19:24

1. The underlying data structure is doubly linked list.
2. The insertion is preserved.
3. Duplicate objects are allowed.
4. Heterogeneous objects are allowed.
5. Null insertion is possible.
6. LinkedList implements Serializable and Cloneable interfaces but it will not implement RandomAccess.
7. LinkedList is the best choice if our frequent operation is insertion or deletion in middle. LinkedList is the worst choice if our frequent operation is retrieval operation.



Constructors:-
LinkedList L = new LinkedList()
LinkedList l = new LinkedList(Collection c); =>Creates an equivalent linked list object for the given collection.

Usually we can use linked list to develop stacks and queues. to provide support for this requirement, linked list class defines the following specific methods:

```
void addFirst(Object o);
void addLast(Object o);
Object getFirst();
Object getLast();
Object removefirst();
Object removeLast();
```

LinkedList → Stack
LinkedList → Queue (CircularQueue)

Difference between ArrayList and LinkedList?

ArrayList	LinkedList
ArrayList a list is the best choice if our frequent operation is retrieval operation.	Linked list is the best choice if our frequent operation is insertion or deletion in the middle.
ArrayList is the worst choice of our frequent insertion or deletion in the middle, because it internally performs several shift operations.	Link list is the worst choice if our frequent operation is retrieval operation.
In addition, the element will be stored in consecutive memory location, and hence retrieval operation will become easy.	In linked list elements won't be stored in consecutive memory location. And hence, retrieval operation is complex or difficult.

```java
package Chapter2;
import java.util.LinkedList;
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList linkList = new LinkedList();
        linkList.add("Raghav");
        linkList.add(20);
        linkList.add(null);
        linkList.add("Shiv");
        System.out.println(linkList);
        linkList.set(0,"Raghav Dhoot");
        System.out.println(linkList);
```

```
        linkList.addFirst(3.1412);
        linkList.addLast("Amitabh Bacchan");
        linkList.addLast("SRK");
        System.out.println(linkList);
        System.out.println(linkList.remove());
        System.out.println(linkList.removeFirst());
        System.out.println(linkList.removeLast());
        System.out.println(linkList);
    }
}
```

Output:

```
[Raghav, 20, null, Shiv]
[Raghav Dhoot, 20, null, Shiv]
[3.1412, Raghav Dhoot, 20, null, Shiv, Amitabh Bacchan, SRK]
3.1412
Raghav Dhoot
SRK
[20, null, Shiv, Amitabh Bacchan]
```

Home-work :- Develop a program to behave like Stack data structure by using LinkedList class of Collection framework.

Develop a program to behave like Queue data structure by using LinkedList class of Collection framework.

Stack

18 February 2025 18:24

1. Child class of Vector class
2. It is specially designed class for LIFO.

```
Stack s = new Stack();
```

Module java.base
Package java.util

Class Stack<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

---

```
public class Stack<E>
extends Vector<E>
```

The `Stack` class represents a last-in-first-out (LIFO) stack of objects. It extends class `Vector` with five operations that allow a vector to be treated as a stack. The usual `push` and `pop` operations are provided, as well as a method to `peek` at the top item on the stack, a method to test for whether the stack is `empty`, and a method to `search` the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example:

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Since:

1.0

Constructors

Constructor	Description
Stack()	Creates an empty Stack.

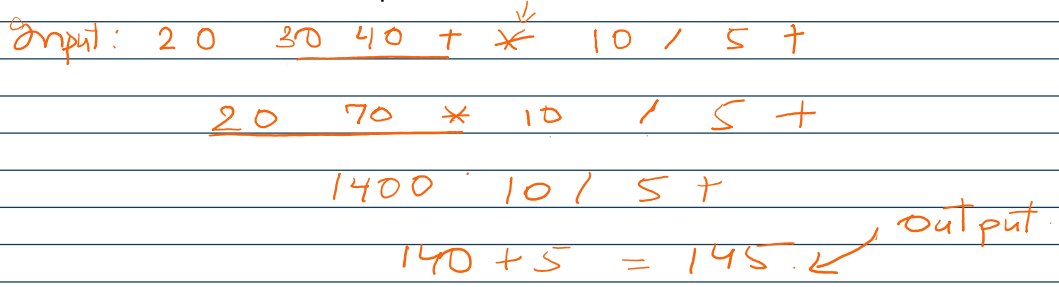
Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
boolean	empty()	Tests if this stack is empty.
E	peek()	Looks at the object at the top of this stack without removing it from the stack.
E	pop()	Removes the object at the top of this stack and returns that object as the value of this function.
E	push(E item)	Pushes an item onto the top of this stack.
int	search(Object o)	Returns the 1-based position where an object is on this stack.

Home work:

WAP to evaluate Postfix expression.



Cursors

18 February 2025 18:37

There are basically 3 types of cursors:

1) Enumeration
2) Iterator
3) ListIterator

If you want to get objects one by one from the Collection then we should go for Cursors.

1. Enumeration: we can use enumeration to get objects one-by-one from legacy collection object. We can create enumeration object by using elements() method of vector class.

```
Enumeration e = v.elements();
```

Methods: public boolean hasMoreElements()
public Object nextElement()

} ← These two methods of Enumeration is implemented in StringTokenizer class

Interface Enumeration<E>

All Known Subinterfaces:
NamingEnumeration<T>

All Known Implementing Classes:
StringTokenizer

public interface **Enumeration<E>**

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series.

For example, to print all elements of a Vector<E> v:

```
for (Enumeration<E> e = v.elements(); e.hasMoreElements();)
    System.out.println(e.nextElement());
```

Methods are provided to enumerate through the elements of a vector, the keys of a hashtable, and the values in a hashtable. Enumerations are also used to specify the input streams to a SequenceInputStream.

NOTE: The functionality of this interface is duplicated by the Iterator interface. In addition, Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

Since:
JDK1.0

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
boolean	hasMoreElements() Tests if this enumeration contains more elements.
E	nextElement() Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

```
package Chapter2;
import java.util.Enumeration;
import java.util.Vector;
public class EnumerationDemo {
    public static void main(String[] args) {
        Vector v = new Vector();
        for(int i = 0; i <=10; i++) {
            v.addElement(i);
        }
        System.out.println(v);
        System.out.println("Let's enumerate:");
        for(Enumeration<Integer> e = v.elements(); e.hasMoreElements();){
            Integer value = (Integer)e.nextElement();
            System.out.println(value);
        }
    }
```

}

Limitations of Enumeration

1. We can apply enumeration concept only for legacy classes. And it is not a universal cursor. And it is not a universal cursor.
2. By using enumeration, we can get only read access. And we cannot perform remove operation.
3. In enumeration, we can move in 1DIRECTION only, that is forward direction.

Iterator interface

We can apply iterator concept for any collection object. And hence, it is a universal cursor.
By using iterator, we can perform both read and remove operations.
We can create Iterator object using iterator() method.

```
Iterator iterator(); of Collection interface

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

Any collection object

1. public boolean hasNext();
2. public Object next();
3. public void remove();

Interface Iterator<E>

Type Parameters:
E - the type of elements returned by this iterator

All Known Subinterfaces:
ListIterator<E>, PrimitiveIterator<T,T_CONS>, PrimitiveIterator.OfDouble, PrimitiveIterator.OfInt, PrimitiveIterator.OfLong, XMLEventReader

All Known Implementing Classes:
BeanContextSupport.BCSIterator, EventReaderDelegate, Scanner

---

public interface **Iterator<E>**

An iterator over a collection. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:
1.2

See Also:
Collection, ListIterator, Iterable

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
default void	forEachRemaining(Consumer<? super E> action) Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
boolean	hasNext() Returns true if the iteration has more elements.
E	next() Returns the next element in the iteration.
default void	remove() Removes from the underlying collection the last element returned by this iterator (optional operation).

Very useful method for iterators in Java.

`Iterator<E>`	`iterator()` Returns an iterator over the elements in this list in proper sequence.
`int`	`lastIndexOf(Object o)` Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`ListIterator<E>`	`listIterator()` Returns a list iterator over the elements in this list (in proper sequence).
`ListIterator<E>`	`listIterator(int index)` Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.

```java
package Chapter2;
import java.util.*;

public class IteratorDemo {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        for(int i = 0; i <= 10; i++) {
            al.add(i);
        }
        System.out.println("ArrayList: " + al);
        Iterator iter = al.iterator();
        while (iter.hasNext()) {
            Integer i = (Integer)iter.next();
            if(i % 2 == 0) {
                System.out.println(i);
            }
            else {
                iter.remove();
            }
        }
        System.out.println("Even arraylist: " + al);
    }
}
```

Output:

```
ArrayList: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
Even arraylist: [0, 2, 4, 6, 8, 10]
```

Limitations of Iterator

- **Forward direction movement is applicable only we cannot move in backward direction. Enumeration and iterator both are single direction cursor.**
- **By using iterator, we can perform only read and remove operation. And we cannot perform replacement and addition of new objects.**

To overcome the second limitation of iterator <u>ListIterator</u> interface is developed.

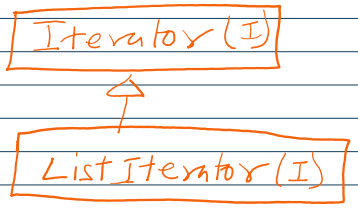
<u>ListIterator</u>

1. By using list iterator, we can move either to the forward direction or to the backward direction And hence, it is a bidirectional cursor.
2. By using list iterator, we can perform replacement and addition of new objects in addition to read and remove operation.

public ListIterator listIterator();

ListIterator liter = obj.listIterator();

Any collection object which has



Any collection object which has List properties.

ListIterator(I)

ListIterator is a child interface of **Iterator** interface. And hence, all methods present in the iterator interface by default available to the ListIterator.

public boolean hasNext()
public Object next()
public int nextIndex()
} Forward movement

public boolean hasPrevious()
public Object previous()
public int previousIndex()
} Backward movement.

Delete
CRUD operations
Create
Read
Update

public void remove();
public void add(Object ob)
public void set(Object ob) ← replaces the current object with ob.
} delete, add, modify or update.

```
package Chapter2;
import java.util.*;
public class LitIteratorDemo {
    public static void main(String[] args) {
        LinkedList liList = new LinkedList();
        liList.add("BalaKrishna");
        liList.add("Rama");
        liList.add("Ravichandran");
        liList.add("LakshmiNarayan");
        liList.add("Bharat");
        System.out.println(liList);
        ListIterator liter = liList.listIterator();
        while (liter.hasNext()) {
            String str = (String)liter.next();
            if(str.equals("Rama")){
                liter.remove();
            }
            else if(str.equals("Bharat")){
                liter.add("Lakshman");
            }
            else if(str.equals("Ravichandran")){
                liter.set("RC");
            }
        }
        System.out.println("List: " + liList);
        while(liter.hasPrevious()){
            String str = (String)liter.previous();
            System.out.println(str);
        }
```

```
    }
}
```

Output:

```
[BalaKrishna, Rama, Ravichandran, LakshmiNarayan, Bharat]
List: [BalaKrishna, RC, LakshmiNarayan, Bharat, Lakshman]
Lakshman
Bharat
LakshmiNarayan
RC
BalaKrishna
```

Note:- The most powerful cursor is ListIterator but its limitation is, it is applicable only for List Objects.

Comparison Table of 3 cursors

Properties	Enumeration	Iterator	List Iterator
Where we can Apply?	Only applicable for legacy classes.	For any Collection Object.	Only for List Objects.
Is it legacy?	Yes (1.0 Version)	No (1.2 Version)	No (1.2 Version)
Movement	Single direction(only forward direction)	Single direction(only forward direction)	Bi-directional
Allowed Operations	Only Read.	Read and remove.	Read, remove, replace, and add new objects.
How we can get object?	By using elements() method of vector class.	By using iterator() of Collection interface.	By using iterator() of List interface.
Methods	2-Methods 1. hasMoreElements() 2. nextElement()	3-methods 1. hasNext() 2. next() 3. remove()	9-methods 1. hasNext() 2. next() 3. nextIndex() 4. hasPrevious() 5. previousIndex() 6. previous() 7. remove() 8. add(Object o) 9. set(Object o)

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	add(E e) Inserts the specified element into the list (optional operation).
boolean	hasNext() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next() Returns the next element in the list and advances the cursor position.
int	nextIndex() Returns the index of the element that would be returned by a subsequent call to next().
E	previous() Returns the previous element in the list and moves the cursor position backwards.
int	previousIndex() Returns the index of the element that would be returned by a subsequent call to previous().
void	remove() Removes from the list the last element that was returned by next() or previous() (optional operation).
void	set(E e) Replaces the last element returned by next() or previous() with the specified element (optional operation).

Internal implementation of Cursor

```java
package Chapter2;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;
public class InternalClassOfCursor {
    public static void main(String[] args) {
        Vector v = new Vector();
        Enumeration e = v.elements();
        Iterator itr = v.listIterator();
        ListIterator litr = v.listIterator();
        System.out.println("\f"+e.getClass().getName());
        System.out.println("\n"+itr.getClass().getName());
        System.out.println("\n"+litr.getClass().getName());
    }
}
```

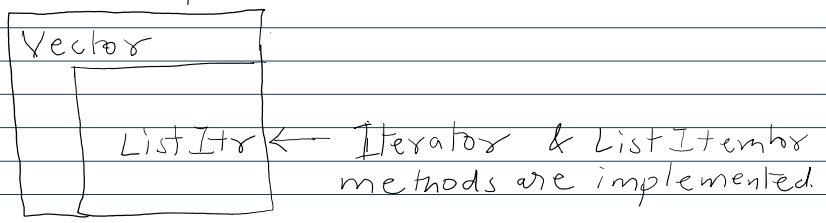
Output:

```
java.util.Vector$1

java.util.Vector$ListItr

java.util.Vector$ListItr
```

anonymous inner class.

Inner class.

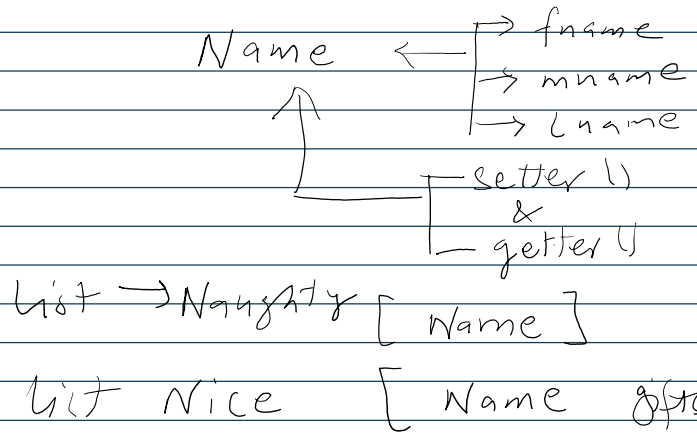
Vector class has implemented all the methods of Enumeration.



Problems

06 March 2025 07:32

Santa Claus allegedly keeps lists of those who are naughty and those who are nice. On the naughty list are the names of those who will get coal in their stockings. On the nice list are those who will receive gifts. Each object in this list contains a name (an instance of Name), and a list of that person's gifts (an instance of an ADT list). Design an ADT for the objects in the nice list. Specify each ADT operation by stating its purpose, by describing its parameters, and by writing preconditions, postconditions, and a pseudocode version of its header. Then write a Java interface for the ADT that includes javadoc-style comments.



Q2: Leetcode, Linklist problem No 430.

Vector class

28 March 2025 09:04

- Underlying data structure is resizable array or growable array.
- Insertion order is preserved.
- Duplicates are allowed.
- Hetrogenous objects are allowed.
- 'null' insertion is possible.
- It implements Serializable, Cloneable, RandomAccess Interfaces.
- Every method present in the Vector is synchronized and hence Vector object is thread safe.

Constructors:

i. Vector v = new Vector(); => Creates an empty vector object with default initial capacity 10. Once vector reaches its maximum capacity, then a new vector object will be created with new capacity = current_capacity*2; (Double capacity.)

ii. Vector v = new Vector(int initialCapacity); => Creates an empty Vector object with specified initial capacity.

iii. Vector v = new Vector(int initialCapacity, int incrementalCapacity);

iv. Vector v = new Vector(Collection c); => Create an equivalent object for the given collection. This constructor is meant for inter conversion between collection object.

Official Documentation:

```
public class Vector<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Each vector tries to optimize storage management by maintaining a `capacity` and a `capacityIncrement`. The `capacity` is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of `capacityIncrement`. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the vector is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The `Enumerations` returned by the `elements` method are *not* fail-fast.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw `ConcurrentModificationException` on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

As of the Java 2 platform v1.2, this class was retrofitted to implement the `List` interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, `Vector` is synchronized. If a thread-safe implementation is not needed, it is recommended to use `ArrayList` in place of `Vector`.

Since:

JDK1.0

Constructor Summary

Constructors

Constructor and Description
`Vector()` Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
`Vector(Collection<? extends E> c)` Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.
`Vector(int initialCapacity)` Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
`Vector(int initialCapacity, int capacityIncrement)` Constructs an empty vector with the specified initial capacity and capacity increment.

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this Vector.
void	`add(int index, E element)` Inserts the specified element at the specified position in this Vector.
boolean	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
boolean	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified Collection into this Vector at the specified position.
void	`addElement(E obj)` Adds the specified component to the end of this vector, increasing its size by one.
int	`capacity()` Returns the current capacity of this vector.
void	`clear()` Removes all of the elements from this Vector.
Object	`clone()` Returns a clone of this vector.
boolean	`contains(Object o)` Returns `true` if this vector contains the specified element.
boolean	`containsAll(Collection<?> c)` Returns true if this Vector contains all of the elements in the specified Collection.
void	`copyInto(Object[] anArray)` Copies the components of this vector into the specified array.
E	`elementAt(int index)` Returns the component at the specified index.
Enumeration<E>	`elements()` Returns an enumeration of the components of this vector.
void	`ensureCapacity(int minCapacity)` Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
boolean	`equals(Object o)` Compares the specified Object with this Vector for equality.
E	`firstElement()` Returns the first component (the item at index `0`) of this vector.
void	`forEach(Consumer<? super E> action)` Performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception.
E	`get(int index)` Returns the element at the specified position in this Vector.
int	`hashCode()` Returns the hash code value for this Vector.
int	`indexOf(Object o)` Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
int	`indexOf(Object o, int index)` Returns the index of the first occurrence of the specified element in this vector, searching forwards from `index`, or returns -1 if the element is not found.
void	`insertElementAt(E obj, int index)` Inserts the specified object as a component in this vector at the specified `index`.
boolean	`isEmpty()` Tests if this vector has no components.

`Iterator<E>`	`iterator()` Returns an iterator over the elements in this list in proper sequence.
`E`	`lastElement()` Returns the last component of the vector.
`int`	`lastIndexOf(Object o)` Returns the index of the last occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
`int`	`lastIndexOf(Object o, int index)` Returns the index of the last occurrence of the specified element in this vector, searching backwards from `index`, or returns -1 if the element is not found.
`ListIterator<E>`	`listIterator()` Returns a list iterator over the elements in this list (in proper sequence).
`ListIterator<E>`	`listIterator(int index)` Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
`E`	`remove(int index)` Removes the element at the specified position in this Vector.
`boolean`	`remove(Object o)` Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
`boolean`	`removeAll(Collection<?> c)` Removes from this Vector all of its elements that are contained in the specified Collection.
`void`	`removeAllElements()` Removes all components from this vector and sets its size to zero.
`boolean`	`removeElement(Object obj)` Removes the first (lowest-indexed) occurrence of the argument from this vector.
`void`	`removeElementAt(int index)` Deletes the component at the specified index.

`boolean`	`removeIf(Predicate<? super E> filter)` Removes all of the elements of this collection that satisfy the given predicate.
`protected void`	`removeRange(int fromIndex, int toIndex)` Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
`void`	`replaceAll(UnaryOperator<E> operator)` Replaces each element of this list with the result of applying the operator to that element.
`boolean`	`retainAll(Collection<?> c)` Retains only the elements in this Vector that are contained in the specified Collection.
`E`	`set(int index, E element)` Replaces the element at the specified position in this Vector with the specified element.
`void`	`setElementAt(E obj, int index)` Sets the component at the specified `index` of this vector to be the specified object.
`void`	`setSize(int newSize)` Sets the size of this vector.
`int`	`size()` Returns the number of components in this vector.
`void`	`sort(Comparator<? super E> c)` Sorts this list according to the order induced by the specified `Comparator`.
`Spliterator<E>`	`spliterator()` Creates a ***late-binding*** and *fail-fast* `Spliterator` over the elements in this list.
`List<E>`	`subList(int fromIndex, int toIndex)` Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
`Object[]`	`toArray()` Returns an array containing all of the elements in this Vector in the correct order.
`<T> T[]`	`toArray(T[] a)`

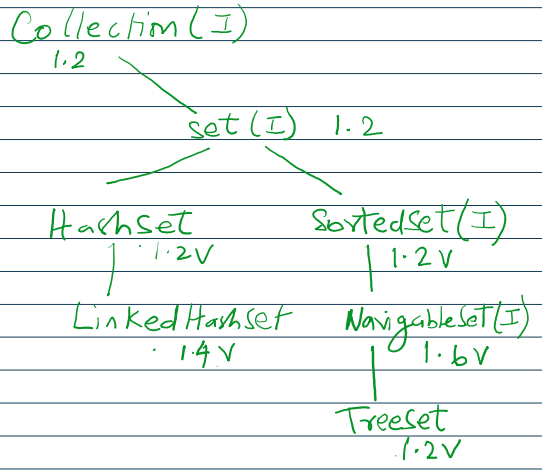
`<T> T[]`	`toArray(T[] a)` Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
`String`	`toString()` Returns a string representation of this Vector, containing the String representation of each element.
`void`	`trimToSize()` Trims the capacity of this vector to be the vector's current size.

Set interface

28 March 2025 09:38

Set is a child interface of Collection. If we want to represent a group of individual object as a single entity where duplicates are not allowed and insertion order is not preserved.

Set interface doesn't contain any new method and have to used Collection interface methods.



Official Documentation

```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the `Set` interface, but they do not contain any additional stipulations.)

The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Some set implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the set may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Since:
1.2

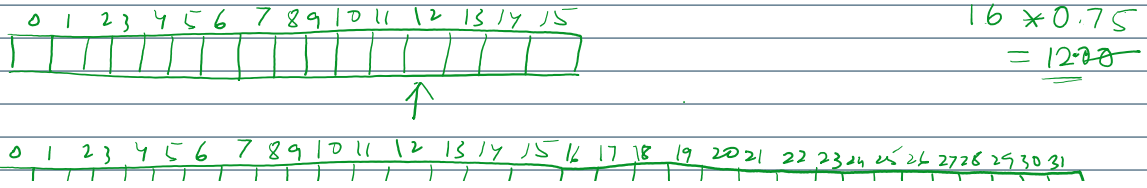
HashSet

- **The underlying data structure is hash table.**
- **Duplicate objects are not allowed.**
- **Insertion order is not preserved.**
- **And it is based on hash code of object.**
- **Null insertion is possible only once.**
- **Heterogeneous objects are allowed.**
- **Implements Serializable and Cloneable interfaces, but not random access interface.**
- **Headset is the best choice if our frequent operation is search operation.**

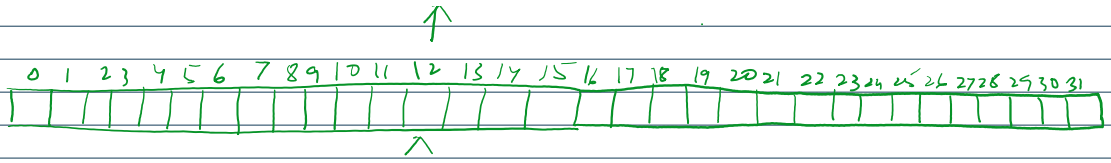
Note: In HashSet duplicates are not allowed. If we trying to insert duplicate then we won't get any compile time or runtime error. And add() method simply returns false.

Constructors:-

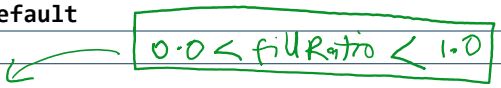
1. HashSet h = new HashSet(); => Creates an empty hash set object with default initial capacity 16 and default fill ratio 0.75.



8
16 * 0.75
= 12



2. HashSet h = new HashSet(int initialCapacity); => Creates an empty hash set object with specified initialCapacity and default fill ratio 0.75.



3. HashSet h = new HashSet(int initialCapacity, float fillRatio);

4. HashSet h = new HashSet(Collection c); => Creates an equivalent HashSet for the given collection. This constructor is meant for inter conversion between Collection object.

Fill Ratio or Load Factor: After filling how much ratio a new hash set object will be created? This ratio is called fill ratio or load factor. For example, fill ratio 0.75 means after filling 75% a new HashSet object will be created.

```java
package Chapter1;
import java.util.*;


/**
 * Write a description of class HashSetDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class HashSetDemo
{
    public static void main(String args[]){
        HashSet h = new HashSet();
        System.out.print("\fHashSet capacity: " + h.size());
        h.add("H");
        h.add("A");
        h.add("S");
        h.add("H");
        h.add("null");
        System.out.print("\n"+h.add("S"));
        h.add("E");
        h.add("T");
        h.add(10);
        h.add(3.14);
        System.out.print("\n" + h);
    }
}
```

Class compiled - no syntax errors

BlueJ: Terminal Window - CollectionFramework

Options

```
HashSet capacity: 0
false
[A, S, null, T, E, 3.14, H, 10]
```

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Note that this implementation is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

The iterators returned by this class's iterator method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*

This class is a member of the Java Collections Framework.

Since:

1.2

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
boolean	add(E e) Adds the specified element to this set if it is not already present.
void	clear() Removes all of the elements from this set.
Object	clone() Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean	contains(Object o) Returns true if this set contains the specified element.
boolean	isEmpty() Returns true if this set contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this set.
boolean	remove(Object o) Removes the specified element from this set if it is present.
int	size() Returns the number of elements in this set (its cardinality).
Spliterator<E>	spliterator() Creates a *late-binding* and *fail-fast* Spliterator over the elements in this set.

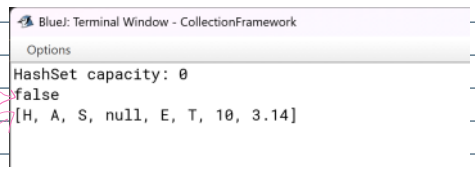
LinkedHashSet

HashSet	LinkedHashSet
Underlying data structure is Hash Table.	Underline data structure is linked list + hash table.
Insertion order is not preserved.	Insertion order is preserved.
Introduced in version 1.2.	Introduced in version 1.4.

LinkedHashSet Is a child class of hash set? It is exactly same as hash said. (including. constructors and methods) except the following difference:

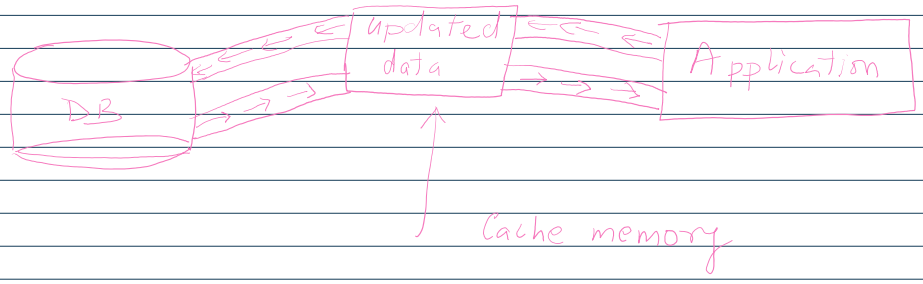
```
package Chapter1;
import java.util.*;


/**
 * Write a description of class HashSetDemo here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class HashSetDemo
{
    public static void main(String args[]){
        LinkedHashSet h = new LinkedHashSet();
        System.out.print("\fHashSet capacity: " + h.size());
        h.add("H");
        h.add("A");
        h.add("S");
        h.add("H");
        h.add("null");
        System.out.print("\n"+h.add("S"));
        h.add("E");
        h.add("T");
        h.add(10);
        h.add(3.14);
        System.out.print("\n" + h);
    }
}
```

Duplicates are not allowed



Insertion order is preserved

In general we can use LinkedHashSet to develop cache based application where duplicates are not allowed and insertion order is preserved



Official Documentation

public class LinkedHashSet<E>
extends HashSet<E>
implements Set<E>, Cloneable, Serializable
Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (*insertion-order*). Note that insertion order is *not* affected if an element is *re-inserted* into the set. (An element e is reinserted into a set s if s.add(e) is invoked when s.contains(e) would return true immediately prior to the invocation.)
This implementation spares its clients from the unspecified, generally chaotic ordering provided by HashSet, without incurring the increased cost associated with TreeSet. It can be used to produce a copy of a set that has the same order as the original, regardless of the original set's implementation:

```
void foo(Set s) {
    Set copy = new LinkedHashSet(s);
    ...
}
```

This technique is particularly useful if a module takes a set on input, copies it, and later returns results whose order is determined by that of the copy. (Clients generally appreciate having things returned in the same order they were presented.)
This class provides all of the optional Set operations, and permits null elements. Like HashSet, it provides constant-time performance for the basic operations (add, contains and remove), assuming the hash function disperses elements properly among the buckets. Performance is likely to be just slightly below that of HashSet, due to the added expense of maintaining the linked list, with one exception: Iteration over a LinkedHashSet requires time proportional to the *size* of the set, regardless of its capacity. Iteration over a HashSet is likely to be more expensive, requiring time proportional to its *capacity*.
A linked hash set has two parameters that affect its performance: *initial capacity* and *load factor*. They are defined precisely as for HashSet. Note, however, that the penalty for choosing an excessively high value for initial capacity is less severe for this class than for HashSet, as iteration times for this class are unaffected by capacity.
Note that this implementation is not synchronized. If multiple threads access a linked hash set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
Set s = Collections.synchronizedSet(new LinkedHashSet(...));
```

The iterators returned by this class's iterator method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-

deterministic behavior at an undetermined time in the future.
Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *the fail-fast behavior of iterators should be used only to detect bugs.*
This class is a member of the Java Collections Framework.
Since:
1.4

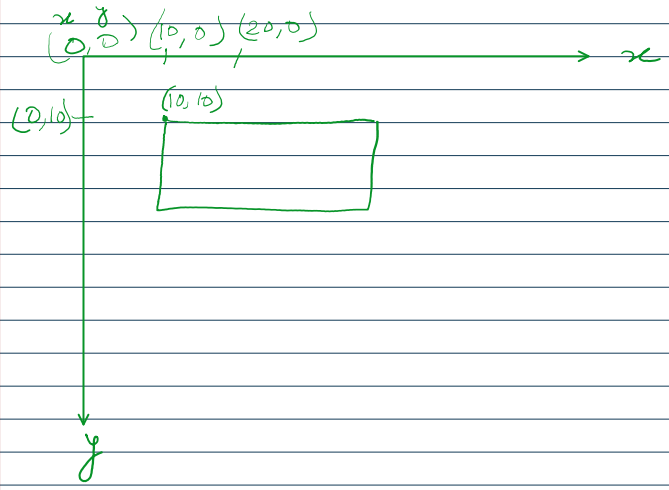
From <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>

Constructor Summary

Constructors

Constructor and Description
`LinkedHashSet()` Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).
`LinkedHashSet(Collection<? extends E> c)` Constructs a new linked hash set with the same elements as the specified collection.
`LinkedHashSet(int initialCapacity)` Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).
`LinkedHashSet(int initialCapacity, float loadFactor)` Constructs a new, empty linked hash set with the specified initial capacity and load factor.

Methods are same as HashSet



SortedSet interface

04 April 2025 12:34

Sorted set is a child interface of Set interface. If we want to represent a group of individual objects, according to some sorting order, without duplicates, then we should go for SortedSet interface.

first() ⇒ 100
last() ⇒ 120
headSet(106) ⇒ [100, 101, 104]
tailSet(106) ⇒ [110, 115, 120]
subSet(101, 115) ⇒ [101, 104, 106, 110]
start ↑ included
end ↑ excluded

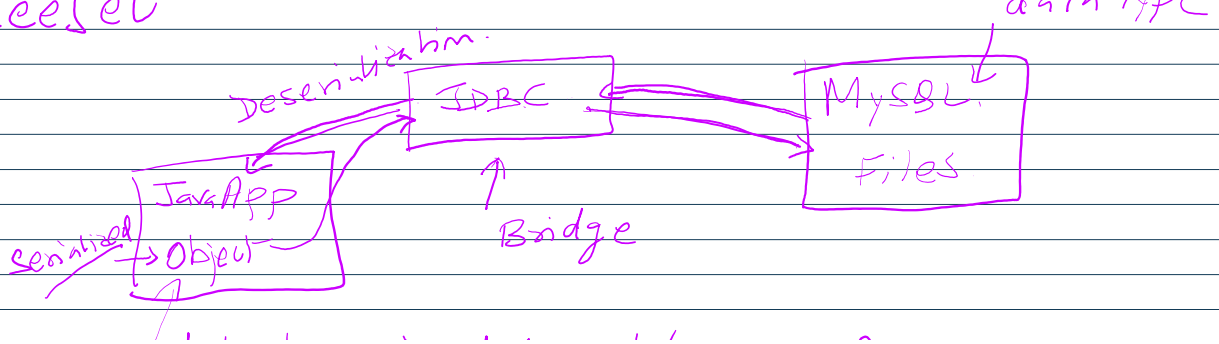
100
101
104
106
110
115
120

Comparator.comparator() ← returns comparator object describes underlying sorting technique. If we are using default natural sorting order, then we will get null.

Note: Default natural sorting order (DNSO) → Ascending order.
For Numbers → Ascending order
Strings → Alphabetical order

Set(I)
↓
SortedSet(I)
↓
→ NavigableSet(I) (1.6 V)
↓
TreeSet ← it implements Serializable, Cloneable but not RandomAccess.

TreeSet



serialize → Object

data type is different from MySQL.

TreeSet

The underlying data structure is balanced tree duplicate objects are not allowed. Insertion order is not preserved. heterogeneous objects are not allowed, otherwise we will get runtime exception saying ClassCastException. 'null' insertion is not possible.
TreeSet implements Serializable, Cloneable but not RandomAccess.
All objects will be inserted based on some sorting order. It may be default natural sorting order or customized sorting order.

Constructors:

TreeSet t = new TreeSet(); => Creates empty TreeSet object, where, although object entered into default natural sorting order.

TreeSet t = new TreeSet(Comparator c); => Creates an empty TreeSet object where the element will be inserted according to customized sorting order specified by the Comparator object.

TreeSet t = new TreeSet(SortedSet s); => Sorted order contained by object s.

TreeSet t = new TreeSet(Collection c); => Default natural sorting order It is an interconversion constructor. Yeah, of course. Inter conversion constructor.

Examples:

```
package Chapter1;
import java.util.TreeSet;
public class TreeSetDemo {
    public static void main(String[] args) {
        TreeSet t = new TreeSet<>();
        t.add("A");
        t.add("a");
        t.add("L");
        t.add("Z");
        t.add("B");
        t.add("C");
        System.out.println(t);
    }
}
```

Output:

```
[A, B, C, L, Z, a]
```

```
package Chapter1;
import java.util.TreeSet;
public class TreeSetDemoUsingStringBuffer {
    public static void main(String[] args) {
        TreeSet t = new TreeSet<>();
        t.add(new StringBuffer("A"));
        t.add(new StringBuffer("a"));
        t.add(new StringBuffer("C"));
        System.out.println(t);
```

```
    }
}
```

Output:

```
[A, C, a]
```

```
package Chapter1;
import java.util.TreeSet;
public class TreeSetDemoUsingStringBuffer {
    public static void main(String[] args) {
        TreeSet t = new TreeSet<>();
        t.add(new StringBuffer("A"));
        t.add(new StringBuffer("a"));
        t.add(new StringBuffer("C"));
        t.add("f");
        System.out.println(t);
    }
}
```

Homogeneous ✓
Comparable ✓
Serializable ✓

Output:

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.StringBuffer cannot be cast to class java.lang.String (
java.lang.StringBuffer and java.lang.String are in module java.base of loader 'bootstrap')
        at java.base/java.lang.String.compareTo(String.java:141)
        at java.base/java.util.TreeMap.put(TreeMap.java:814)
        at java.base/java.util.TreeMap.put(TreeMap.java:534)
        at java.base/java.util.TreeSet.add(TreeSet.java:255)
        at Chapter1.TreeSetDemoUsingStringBuffer.main(TreeSetDemoUsingStringBuffer.java:12)
PS C:\Users\bluej\Documents\Students\RaghavISC\Java4Sem\Coding>
```

Comparable interface & Comparator Interface

14 April 2025 19:00

It is present in Java.lang package and it can contains only one method. compareTo():

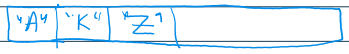
```
public int compareTo(Object o);
```

Obj1.compareTo(obj2)

- → returns -ve iff obj1 has to come before obj2.
- → returns -ve iff obj1 has to come after obj2
- → return 0 iff obj1 and obj2 are equal.

"A".compareTo("Z") ⇒ -ve
"Z".compareTo("A") ⇒ +ve
"A".compareTo("A") ⇒ 0
"A".compareTo(null) ⇒ NullPointerException

```
TreeSet t = new TreeSet();
t.add("K");
t.add("Z");
t.add("A");
```



"Z".compareTo("K") +ve
"A".compareTo("Z") -ve
again, "A".compareTo("K") -ve

If we are depending on "default natural sorting order", then while adding objects into the TreeSet, JVM will call compareTo() method.

Note: If DNSO is not available or if we are not satisfied with DNSO then we can go for customized sorting by using Comparator(I).

Comparable(I) is ment for DNSO, whereas Comparator(I) is ment for customized sorting order.

Comparator (I)

Present in java.util package. And, it define two abstract methods.

① compare() ② equals()

① public int compare(Object obj, Object obj2)

- → returns -ve iff obj1 has to come before obj2.

→ returns -ve iff obj1 has to come before obj2.
→ returns -ve iff obj1 has to come after obj2
→ return 0 iff obj1 and obj2 are equal

(2) public boolean equals(Object obj)

Wherever we are implementing comparator interface, compulsory we should provide only. compare() method and we are not required to provide equals() method because it is already available to our class from object class through inheritance.

```
public class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2){
        - - - - -
    }
}
```

```
package Chapter1;
import java.util.Comparator;
public class MyComparator implements
Comparator{
    @Override
    public int compare(Object ob1, Object ob2){
        Integer i1 = (Integer)ob1;
        Integer i2 = (Integer)ob2;
        return (i1 < i2)?1:(i1 > i2)? -1: 0;
    }
}
```

← logic of descending

```
package Chapter1;
import java.util.TreeSet;
public class IntegerTreeSetDemo {
    public static void main(String[] args) {
        MyComparator c = new MyComparator();
        TreeSet t = new TreeSet<>(c);
        t.add(10);
        t.add(-1);
        t.add(15);
        t.add(17);
        t.add(1);
        t.add(5);
        t.add(25);
        System.out.println(t);
    }
}
```

Comparator object is passed here.

← here compare function is called automatically for comparison of value.

Output:

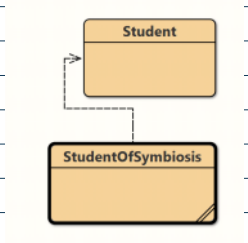
```
[25, 17, 15, 10, 5, 1, -1]
```

Comparable (I) vs Comparator (I)

1. For predefined comparable class (DNSO) is already available. If we are not satisfied with that DNSO then we can define our own sorting by using Comparator (I).
2. For predefined non-comparable classes DNSO not already available so we can define our own sorting by using Comparator (I).
3. For our own classes like Student, the person who is writing the class is responsible to define, DNSO by implementing Comparable (I). The person who is using our class, if he is not satisfied with DNSO then he can define his own sorting order by using Comparator(I).

Student st1, st2

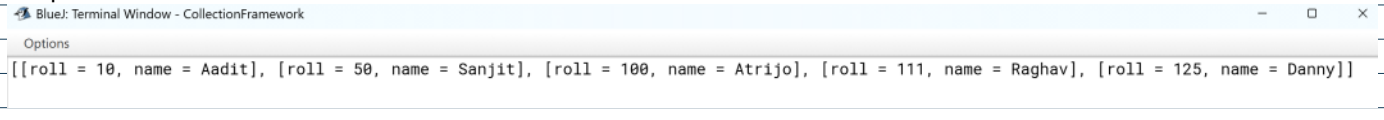
st1.compareTo(st2) = -ve or +ve

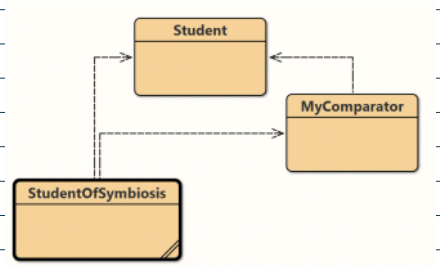



```
package ComparableComparator;


/**
 * Write a description of class Student here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Student implements Comparable
{
    private int roll;
    private String name;

    public Student(int roll, String name){
        this.roll = roll;
        this.name = name;
    }
    public int getRoll(){
        return this.roll;
    }
```

```
    public String getName(){
        return this.name;
    }
    @Override
    public int compareTo(Object o){

        Student st = (Student)o;
        if(this.roll < st.roll){
            return -1;
        }
        else if(this.roll > st.roll){
            return 1;
        }
        else{
            return 0;
        }
    }

    @Override
    public String toString(){
        return "[roll = " + roll + ", name = " + name + "]";
    }
}


package ComparableComparator;
import java.util.TreeSet;


/**
 * Write a description of class StudentOfSymbiosis here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class StudentOfSymbiosis
{
    public static void main(String args[]){
        Student st1 = new Student(100, "Atrijo");
        Student st2 = new Student(10, "Aadit");
        Student st3 = new Student(50, "Sanjit");
        Student st4 = new Student(125, "Danny");
        Student st5 = new Student(111, "Raghav");

        TreeSet t = new TreeSet();
        t.add(st1);
        t.add(st2);
        t.add(st3);
        t.add(st4);
        t.add(st5);

        System.out.print("\f" + t);
    }
}
```

Output:





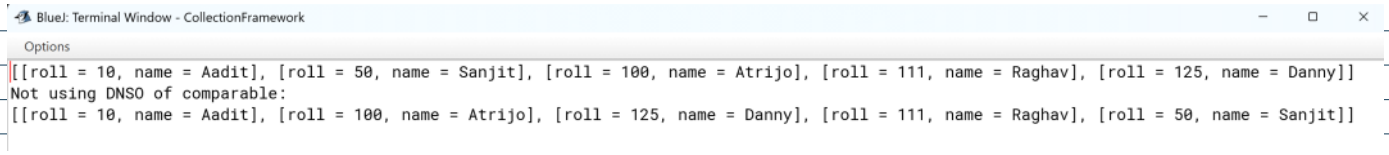
Student class is defined above.

```
package ComparableComparator;
import java.util.Comparator;


/**
 * Write a description of class MyComparator here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class MyComparator implements Comparator
{
    @Override
    public int compare(Object o1, Object o2){
        Student s1 = (Student)o1;
        Student s2 = (Student)o2;
        return s1.getName().compareTo(s2.getName());
    }
}
```

```
package ComparableComparator;
import java.util.TreeSet;


/**
 * Write a description of class StudentOfSymbiosis here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class StudentOfSymbiosis
{
    public static void main(String args[]){
        Student st1 = new Student(100, "Atrijo");
        Student st2 = new Student(10, "Aadit");
        Student st3 = new Student(50, "Sanjit");
        Student st4 = new Student(125, "Danny");
        Student st5 = new Student(111, "Raghav");

        TreeSet t = new TreeSet();
        t.add(st1);
        t.add(st2);
        t.add(st3);
        t.add(st4);
        t.add(st5);

        System.out.print("\f" + t);

        TreeSet t1 = new TreeSet(new MyComparator());
        t1.add(st1);
        t1.add(st2);
        t1.add(st3);
```

```
        t1.add(st4);
        t1.add(st5);

        System.out.print("\nNot using DNSO of comparable:");
        System.out.print("\n" + t1);
    }
}
```

Output:



Comparison of Set implemented Classes

Property	HashSet	LinkedHashSet	TreeSet
Underlying data structure.	Hash Table	LinkedList+HashTable	BalancedTree
Duplicate Object	Not Allowed	Not Allowed	Not Allowed
Insertion Order	Not Preserved	Not Preserved	Not Preserved
Sorting Order	Not Applicable	Not Applicable	Applicable
Hetrogenous Object	Allowed	Allowed	Not Allowed
Null Acceptance	Allowed(only once)	Allowed(only once)	Not allowed after 1.7V