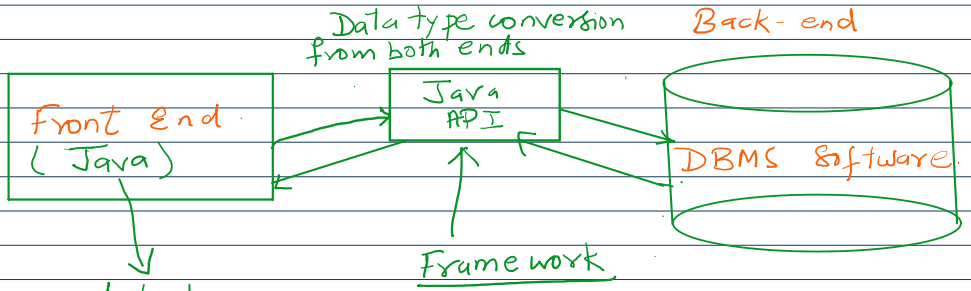
Collection (API)

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.

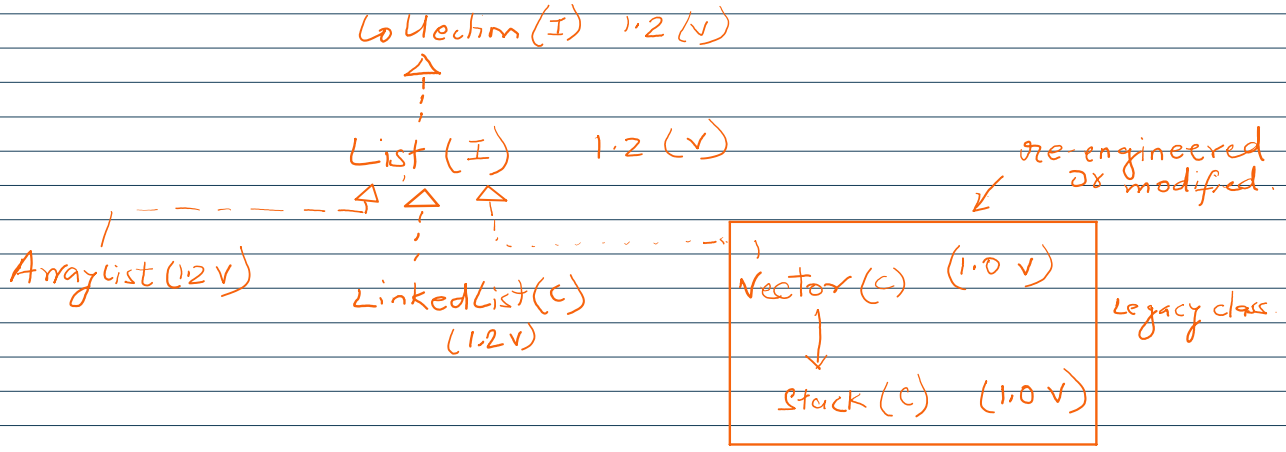
1. Collections 2. List 3. Set 4. SortedSet
5. NavigableSet 6. Queue 7. Map 8. SortedMap
9. NavigableMap

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

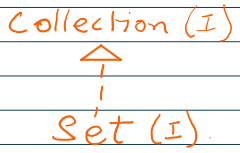
entity, then we should go for Collection interface.

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

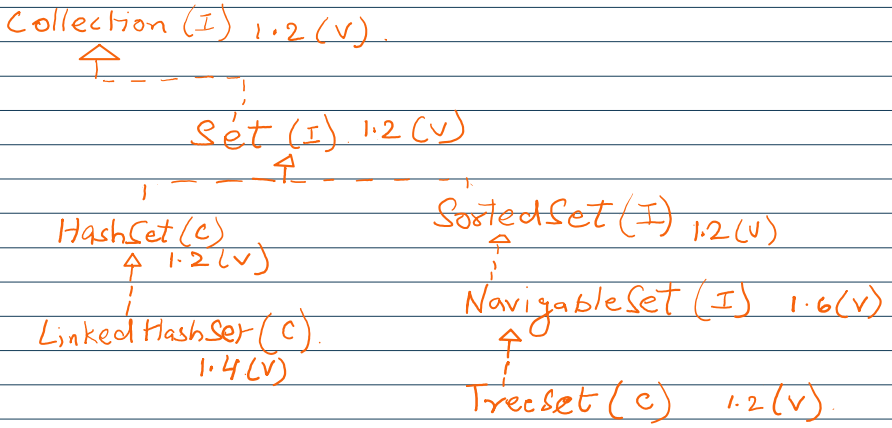


List(I): It is a child interface of Collection(I). If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order must be preserved, then we should go for list interface.

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.

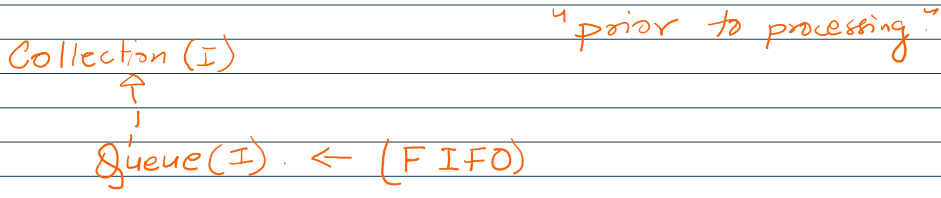


Sorted Set(I): It is the child interface of **Set** interface. If we want to represent a group of individual object as a single entity, where it duplicates are not allowed, and all objects should be inserted according to some sorted order, then we should go for **SortedSet** interface.

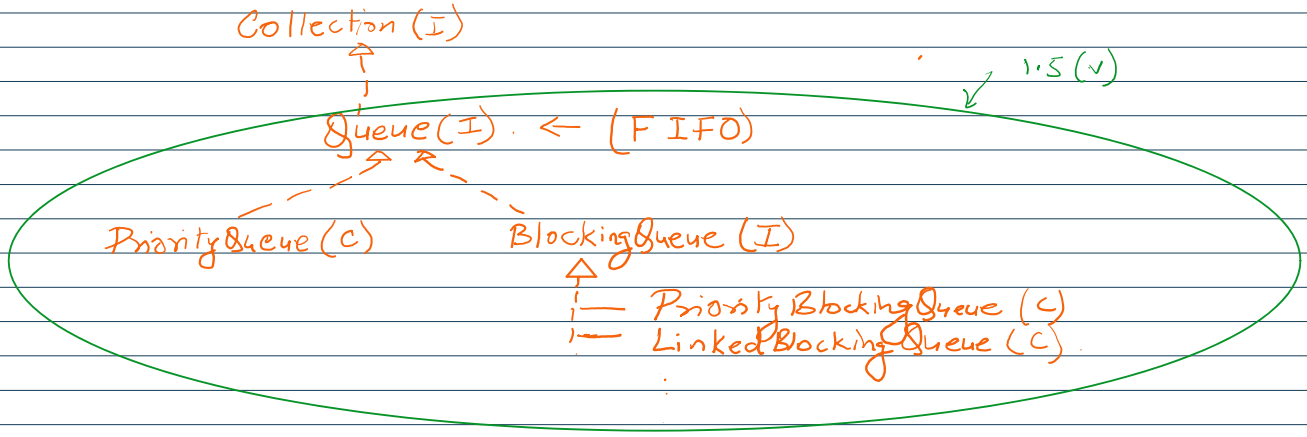
NavigableSet(I): It is a child interface of **SortedSet** interface. It contains several methods for navigation purposes (Scrolling of data).

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.



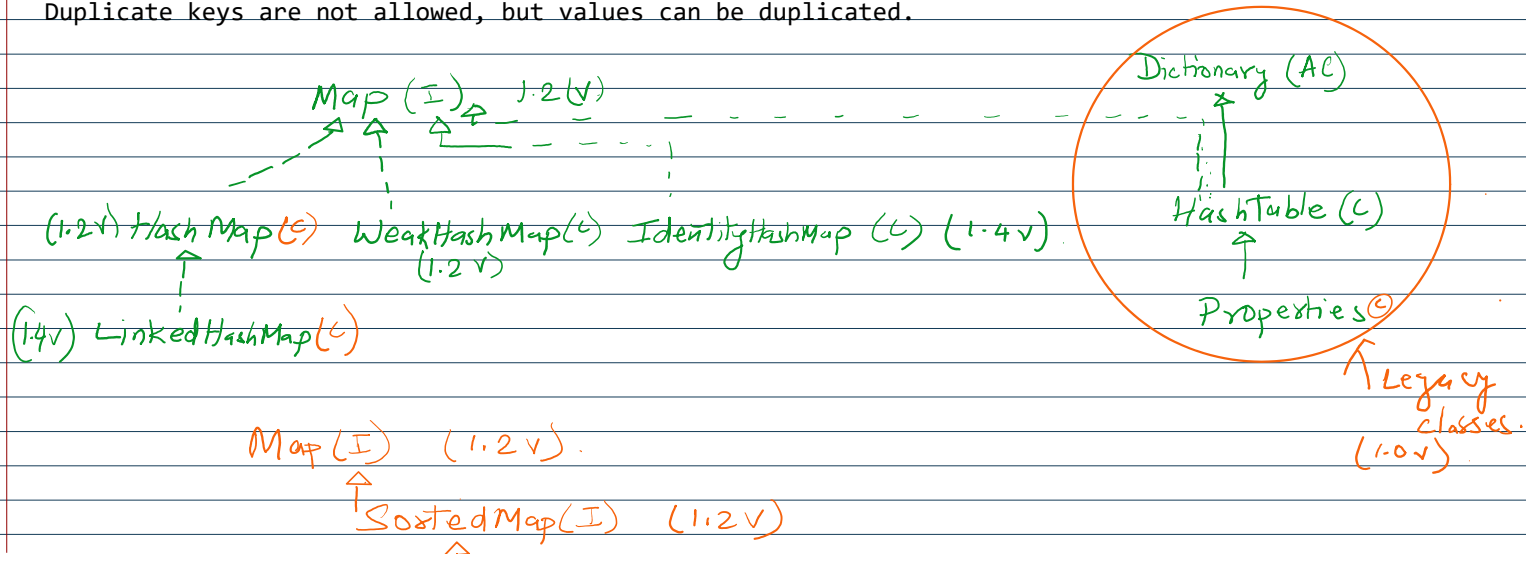
Queue(I): Queue is a child interface of collection interface. If we want to represent a group of individual "prior to processing" then we should go for Queue. Usually queue follows FIFI order. But based on our requirement, we can implement our own priority order also. Example, before sending a mail, 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.

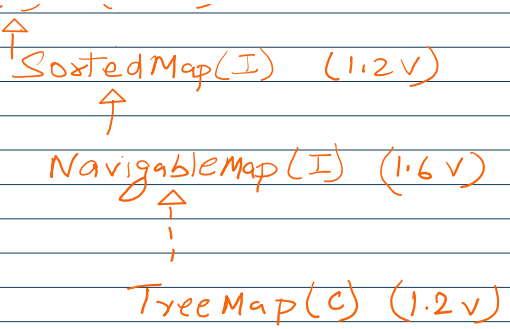


Note: All the above interfaces (Collection, List, Set, SortedSet, NavigableSet. Queue) ment for representing a group of individual objects.

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

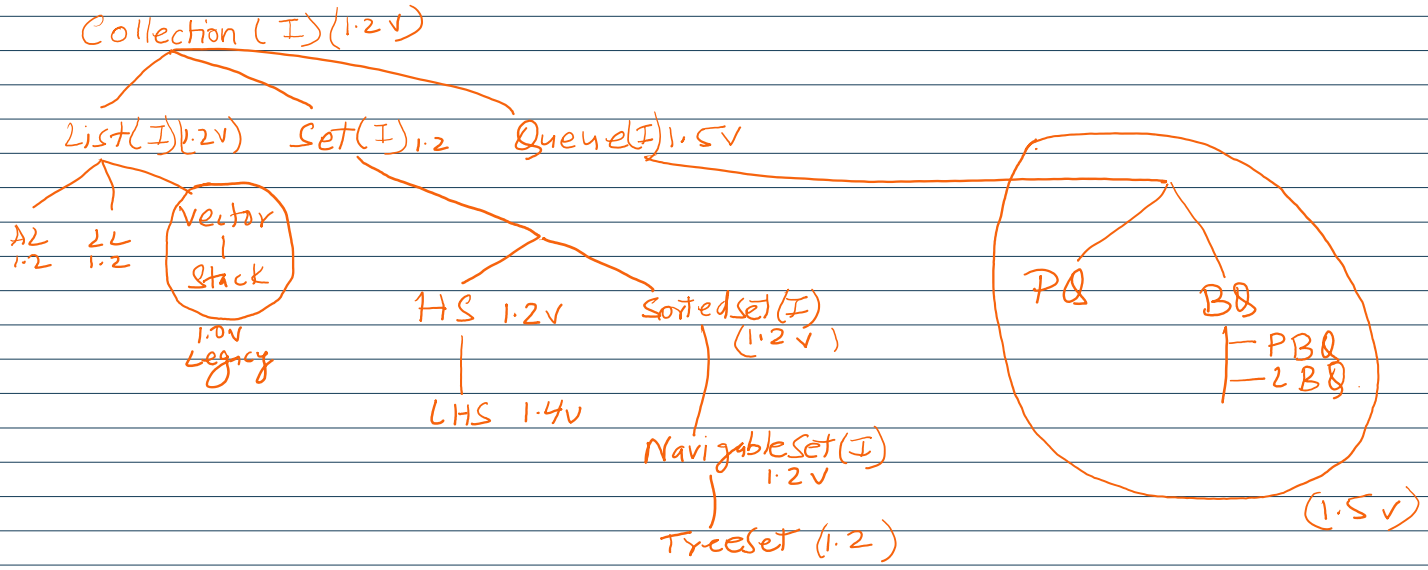
Map(I): Map is not child interface of collection. If we want to represent a group of object as key value pairs, then we should go for map interface. both key and value are objects only. Duplicate keys are not allowed, but values can be duplicated.

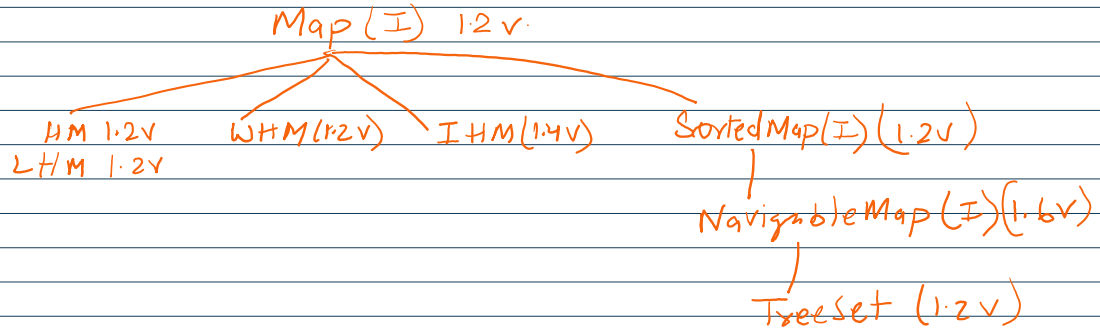




SortedMap(I): It is a child interface of **Map** interface. If we want to represent a group of key-value pairs according to some sorting order of keys, then we should go for **SortedMap** interface. In sorted map interface, the shorting should be based on key, but not based on value.

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

Concept of Generics <E> <?> <super ?>

<T>

Collection interface

28 January 2025 20:21

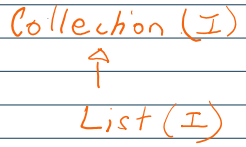
If we want to represent a group of individual objects as a single entity, then we should go for Collection.
Collection interface defines the most common methods which are applicable for any Collection object.

```
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.

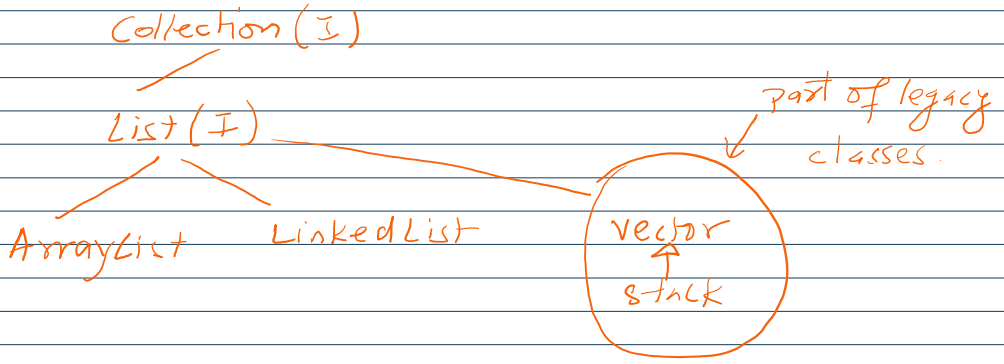


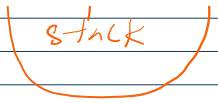
List interface:

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

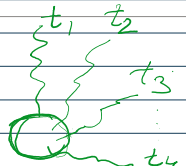
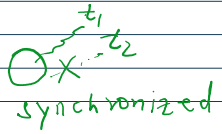
List has implemented
Linked ⇒ 1. Serializable
⇒ 2. Cloneable
it hasn't implemented RandomAccess

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();
```

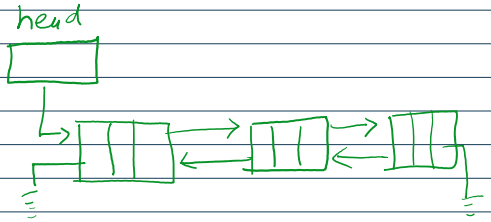
```java
package Chapter1;
import java.util.Vector;


/**
 * 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.

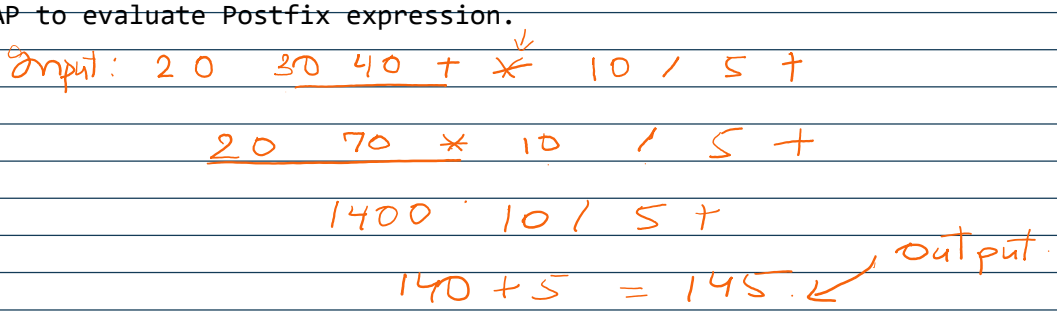
Method Summary

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.

```java
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.

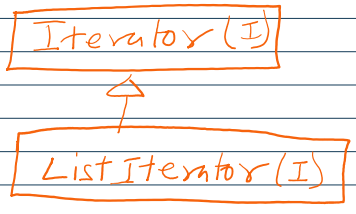
<u>ListIterator</u>

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.

```
public void remove();
public void add(Object ob)
public void set(Object ob)
```
} delete, add, modify or update.

set(Object ob) ← replaces the current object with ob.

Delete

C R U D operations

Create

Read

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());
    }
}
```

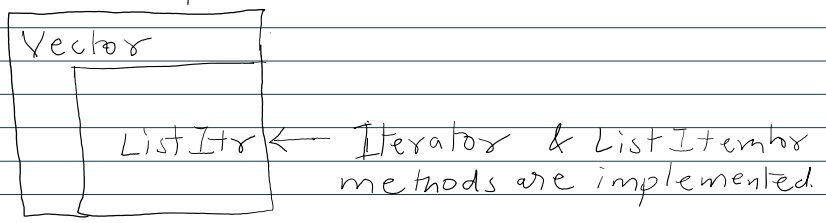
Output:

```
java.util.Vector$1

java.util.Vector$ListItr

java.util.Vector$ListItr
```

anonymous inner class.

Inner class.

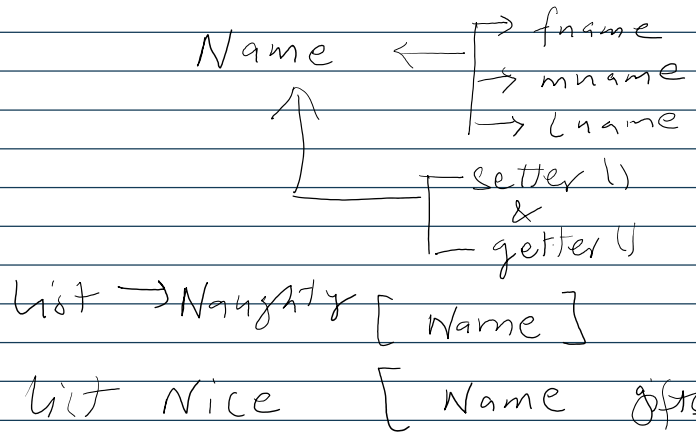
Vector class has implemented all the methods of Enumeration.



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)`**

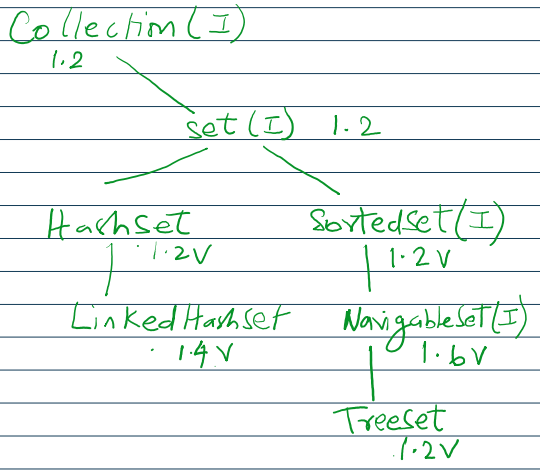
`<T> T[]`	**`toArray(T[] a)`** Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
`String`	**`toString()`** Returns a string representation of this Vector, containing the String representation of 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

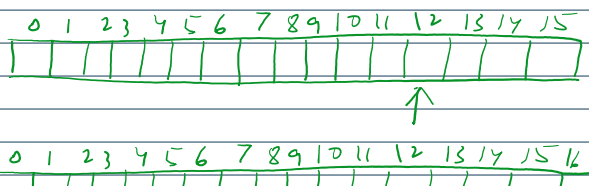
HashSet

- **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.**

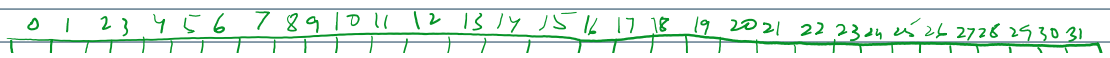
Note: In HashSet duplicates are not allowed. If we trying to insert duplicate then we won't get any compile time or runtime error. And add() method simply returns false.

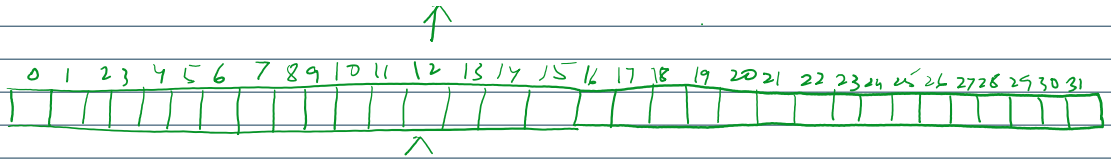
Constructors:-

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

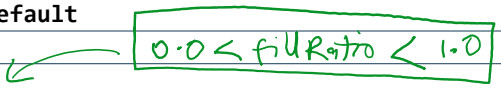


8
16 * 0.75
= 12





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.

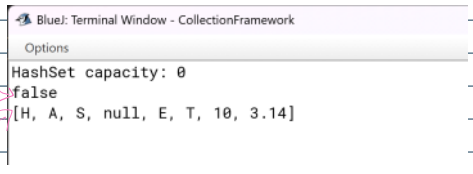
LinkedHashSet

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:

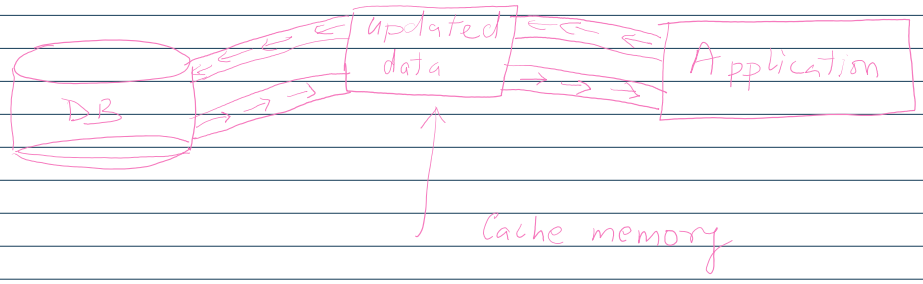
```
package Chapter1;
import java.util.*;


/**
 * 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

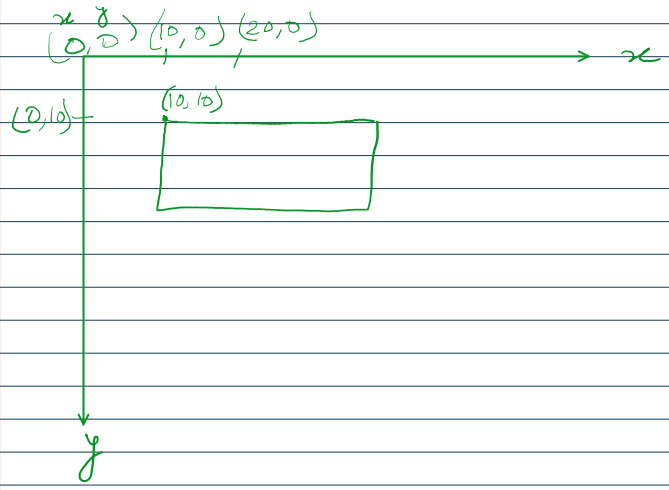
From <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>

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

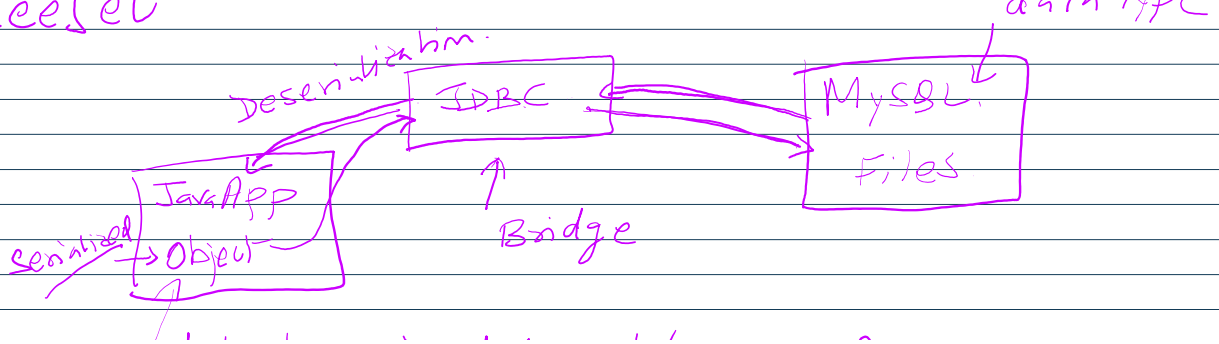
100
101
104
106
110
115
120

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():

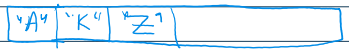
```
public int compareTo(Object o);
```

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:

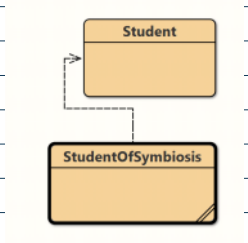
```
[25, 17, 15, 10, 5, 1, -1]
```

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

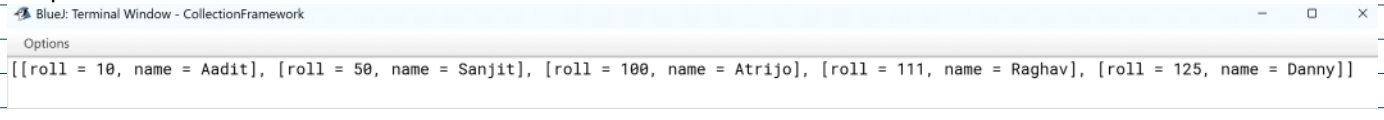
st1.compareTo(st2) = -ve or +ve

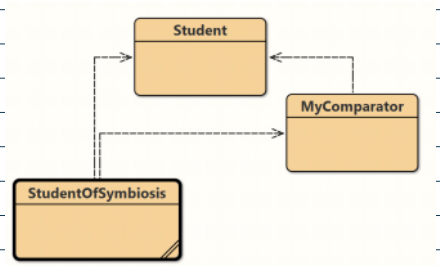



```
package ComparableComparator;


/**
 * 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.

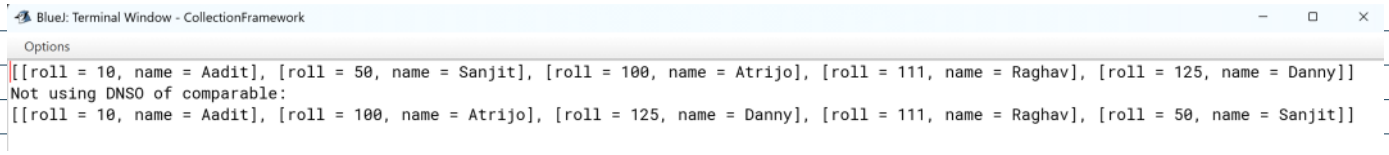
```
package ComparableComparator;
import java.util.Comparator;


/**
 * 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);
```

Time to sleep in the night
Take bath by (7:30 to 7:45) AM

```
        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

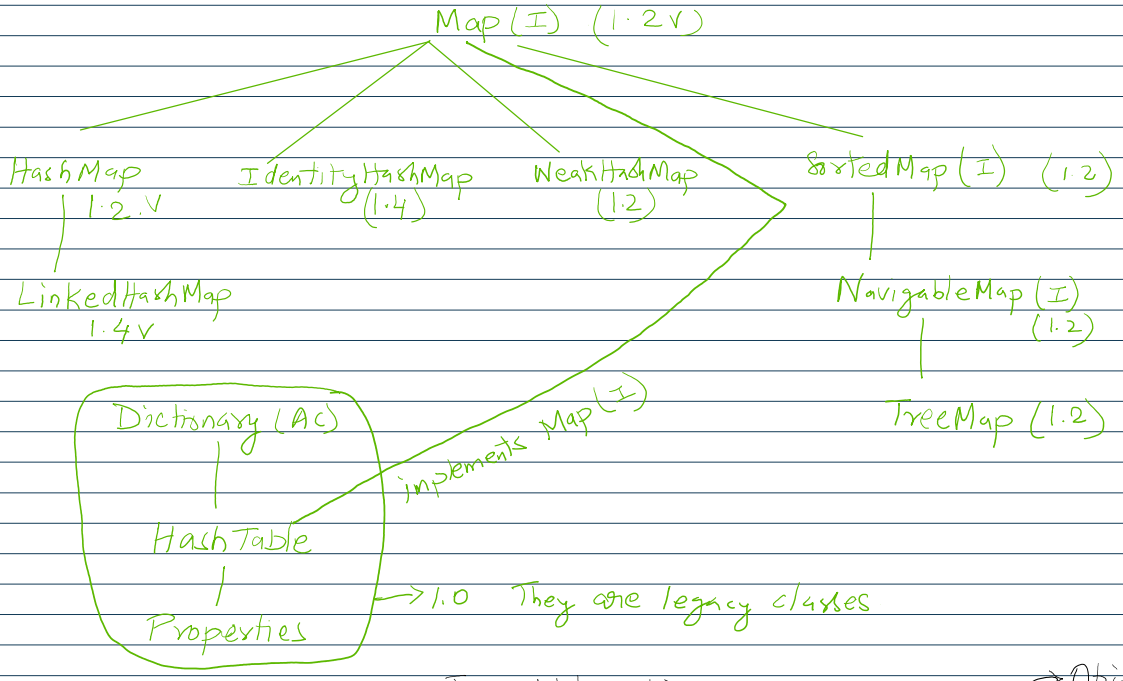
The most powerful cursor is ListIterator but its Limitation is, it is applicable only for List Objects.

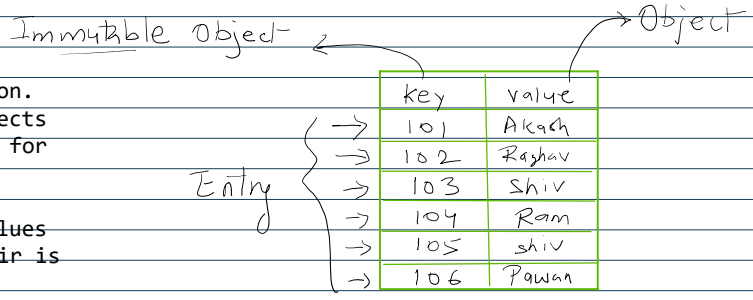
Comparison table of 3 Cursors

Properties	Enumeration	Iterator	ListIterator
Where we can apply?	Only for legacy classes.	For any Collection Object.	Only for List Object.
Is it legacy	Yes (1.0)	No(1.2)	No(1.2)
Movement	Single direction (only forward direction).	Single direction (only forward direction).	Bidirectional
Allowed operations	Only read.	Read and remove.	Read, remove, replace, add
How we can get the object?	By using elements() method of Vector class.	By using iterator() method of Collection interface.	By using listIterator() method of List(I)
Methods	2-methods	3-methods	9-methods.

Map interface

05 May 2025 19:32





- Map is not child interface of collection.
- If we want to represent a group of objects as key - value pair, then we should go for map.
- Both keys and values are objects only.
- Duplicate keys are not allowed. but values can be duplicated. Each key - value pair is called Entry.
- Hence map is considered as a collection of Entry objects.

Primary key Attribute : rest of attributes

key : value

Map (I) methods

Object put (Object key, Object value)

old value

m.put (101, 'Arjun')

null

m.put (102, 'Ravi')

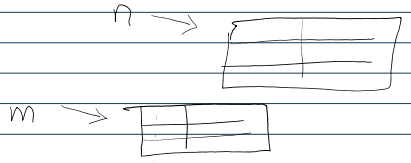
null

m.put (101, 'Krishna')

Arjun

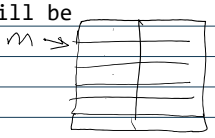
JDBC

101	~~Arjun~~ Krishna
102	Ravi



To add one key value pair to the map. If the key is already present, then old value will be replaced with new value and returns old value.

m.putAll (n);



Object putAll (Map m)

Object get (Object key) → returns the value associated with specified key

Object remove (Object key) → removes the entry associated with specified key and it returns the value.

boolean containsKey (Object key)

boolean containsKey(Object key)
boolean containsValue(Object value)
boolean isEmpty()
int size()
void clear()
Set keySet()
Collection values()
Set entrySet()
} Collection views of Map

Entry(I) is an inner interface

```
interface Map{
    interface Entry{
        Object get();
        Object getValue();
        Object setValue();
    }
}
```

} Entry Specific Method - We can apply only on Entry objects

```
Set s = h.entrySet();
for(Iterator i : s.listIterator())
Map.Entry ob = i.get();
```

Iterable object

interface

A Map is a group of key-value pair, and each key-value pair is called an Entry interface. Hence Map is considered as a Collection of Entry objects. Without existing Map object, there is no chance of existing Entry object. Hence, Entry interface is defined inside map interface.

Official Documentation:

public interface Map<K,V>

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.
The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their encounter order; others, like the HashMap class, do not. Maps with a defined encounter order are generally subtypes of the SequencedMap interface.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void

(no arguments) constructor which creates an empty map, and a constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.
The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw UnsupportedOperationException if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the map. For example, invoking the putAll(Map) method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.
Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible key or value 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 key or value whose completion would not result in the insertion of an ineligible element into the map 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.
Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the containsKey(Object key) method says: "returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))." This specification should *not* be construed to imply that invoking Map.containsKey with a non-null argument key will cause key.equals(k) to be invoked for any key k. Implementations are free to implement optimizations whereby the equals invocation is avoided, for example, by first comparing the hash codes of the two keys. (The Object.hashCode() specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying Object methods wherever the implementor deems it appropriate.
Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the clone(), equals(), hashCode() and toString() methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.
Unmodifiable Maps
The Map.of, Map.ofEntries, and Map.copyOf static factory methods provide a convenient way to create unmodifiable maps. The Map instances created by these methods have the following characteristics:

- They are *unmodifiable*. Keys and values cannot be added, removed, or updated. Calling any mutator method on the Map will always cause UnsupportedOperationException to be thrown. However, if the contained keys or values are themselves mutable, this may cause the Map to behave inconsistently or its contents to appear to change.
- They disallow null keys and values. Attempts to create them with null keys or values result in NullPointerException.
- They are serializable if all keys and values are serializable.
- They reject duplicate keys at creation time. Duplicate keys passed to a static factory method result in IllegalArgumentException.
- The iteration order of mappings is unspecified and is subject to change.
- They are value-based. Programmers should treat instances that are equal as interchangeable and should not use them for synchronization, or unpredictable behavior may occur. For example, in a future release, synchronization may fail. Callers should make no assumptions about the identity of the returned instances. Factories are free to create new instances or reuse existing ones.
- They are serialized as specified on the Serialized Form page.

This interface is a member of the Java Collections Framework.
Since:
1.2
https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Map.html for methods in Map(I).

HashMap

- **The underlying data structure is HashTable.**
- **Insertion order is not preserved and it is based on hash code of key.**
- **Duplicate keys are not allowed, but values can be duplicated.**
- **Heterogeneous objects are allowed for both key and value.**
- **"null" is allowed for key (only once) Null is allowed for values (any number of times).**
- **HashMap implements Map<k><v>, Serializable, Cloneable interfaces but not RandomAccess interface.**
- **Hash map is the best choice if our frequent operation is search operation.**

Constructors:

HashMap m = new HashMap(); ==> Creates an empty hash map object with default initial capacity. 16 and default fill ratio, 0.75

HashMap m = new HashMap(int initialCapacity); ==> Creates empty hash map object with a specified initial capacity and default filled ratio, 0.75.

HashMap m = new HashMap(int initialCapacity, float fillRatio); ==> Creates empty hashmap object with specified initial capacity and specified fill ratio.
The value of filled ratio or load factor is: [0.0 < fill ratio <= 1.0]

HashMap m = new HashMap(Map m);

Official Documentation:

public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.
An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.
As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.
If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table. Note that using many keys with the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.
Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally 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:
1.2

From <https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/HashMap.html>

HashMap.entrySet() : Set
Returns a Set view of the mappings contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own remove operation, or through the setValue operation on a map entry returned by the iterator) the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.
Specified by: entrySet() in Map, **Overrides:** entrySet() in AbstractMap

- **Returns:**
  - a set view of the mappings contained in this map

Demo code:

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashMapDemo {
    public static void main(String[] args) {
        HashMap m = new HashMap<>();
        m.put("Chrinjeevi", "1000");
        m.put("Venketesh",800);
        m.put("Nagarjuna", 500);
        m.put("Ramesh Babu", 900);
        m.put("Rajanikant", 1500);
        System.out.println("\n" + m);
        Set s = m.keySet();
        System.out.println("keys: "+s);
        Collection c = m.values();
        System.out.println("Values: " + c);
        Set s1 = m.entrySet();
        System.out.println("Entery Set: " + s1 );
        Iterator itr = s1.iterator();
        while (itr.hasNext()) {
            Map.Entry m1 = (Map.Entry)itr.next();
            System.out.println(m1.getKey()+"===>" + m1.getValue());
            String key = (String)m1.getKey();
            if(key.equals("Nagarjuna")){
                m1.setValue(10000);
            }
        }
        System.out.println(m);
    }
}
```

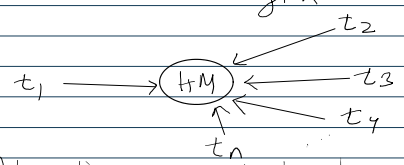
Output:

```
{Ramesh Babu=900, Rajanikant=1500, Venketesh=800, Chrinjeevi=1000, Nagarjuna=500}
keys: [Ramesh Babu, Rajanikant, Venketesh, Chrinjeevi, Nagarjuna]
Values: [900, 1500, 800, 1000, 500]
Entery Set: [Ramesh Babu=900, Rajanikant=1500, Venketesh=800, Chrinjeevi=1000, Nagarjuna=500]
Ramesh Babu===>900
Rajanikant===>1500
Venketesh===>800
Chrinjeevi===>1000
Nagarjuna===>500
{Ramesh Babu=900, Rajanikant=1500, Venketesh=800, Chrinjeevi=1000, Nagarjuna=10000}
```

Difference between HashMap and HashTable

HashMap	HashTable
① Not Synchronized: Every method in HashMap is not synchronized.	① Synchronized: Every method present in the HashTable is synchronized.

① Not Synchronized: Every method in HashMap is not synchronized.



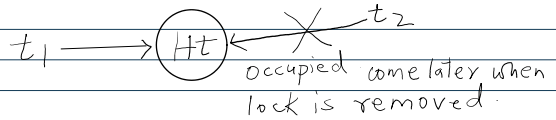
At a time multiple threads are allowed to operate on HashMap object and hence it is not a Thread safe.

2. Relatively performance is high because threads are not required to wait to operate on HashMap object.

3. 'null' is allowed for key (only once)
'null' is allowed for value (multiple times).

4. It is not a legacy, it is introduced in 1.2V.

① Synchronized: Every method present in the HashTable is synchronized.



At a time only one thread is allowed to operate on HashTable and hence it is Thread Safe.

2. Relatively performance is low because threads are required to wait to operate on HashTable object.

3. 'null' such type of concept is not available if given, then NullPointerException will occur.

4. It is a legacy, introduced in 1.0V

How to get synchronized version of HashMap object?

```
HashMap m = new HashMap();
Map m1 = Collections.synchronizedMap(m);
```

m1 → Synchronized.
m → not synchronized

By default HashMap is non-synchronized but we can get synchronized version of HashMap by using synchronizedMap() method of Collections class.

HashMap ↑ LinkedHashMap

HashSet ↑ LinkedHashSet

It is a child class of HashMap. It is exactly same as HashMap (including method and constructor), except the following differences)

HashMap	LinkedHashMap
The underlying data structure is HashTable.	The underlying data structure is combination of LinkedList and HashTable.(Hybrid Data structure)
Insertion order is not preserved and it is based on hash code of keys.	Insertion order is preserved.
Introduced in 1.2 version.	Introduced in 1.4 version.

Demo Code:

```
import java.util.*;
public class LinkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap m = new LinkedHashMap(); ← Here, we have changed HashMap to LinkedHashMap.
        m.put("Chrinjeevi", "1000");
        m.put("Venketesh",800);
        m.put("Nagarjuna", 500);
        m.put("Ramesh Babu", 900);
        m.put("Rajanikant", 1500);
```

```
        System.out.println("\n" + m);
        Set s = m.keySet();
        System.out.println("keys: "+s);
        Collection c = m.values();
        System.out.println("Values: " + c);
        Set s1 = m.entrySet();
        System.out.println("Entery Set: " + s1 );
        Iterator itr = s1.iterator();
        while (itr.hasNext()) {
            Map.Entry m1 = (Map.Entry)itr.next();
            System.out.println(m1.getKey()+"===>" + m1.getValue());
            String key = (String)m1.getKey();
            if(key.equals("Nagarjuna")){
                m1.setValue(10000);
            }
        }
        System.out.println(m);
    }
}
```

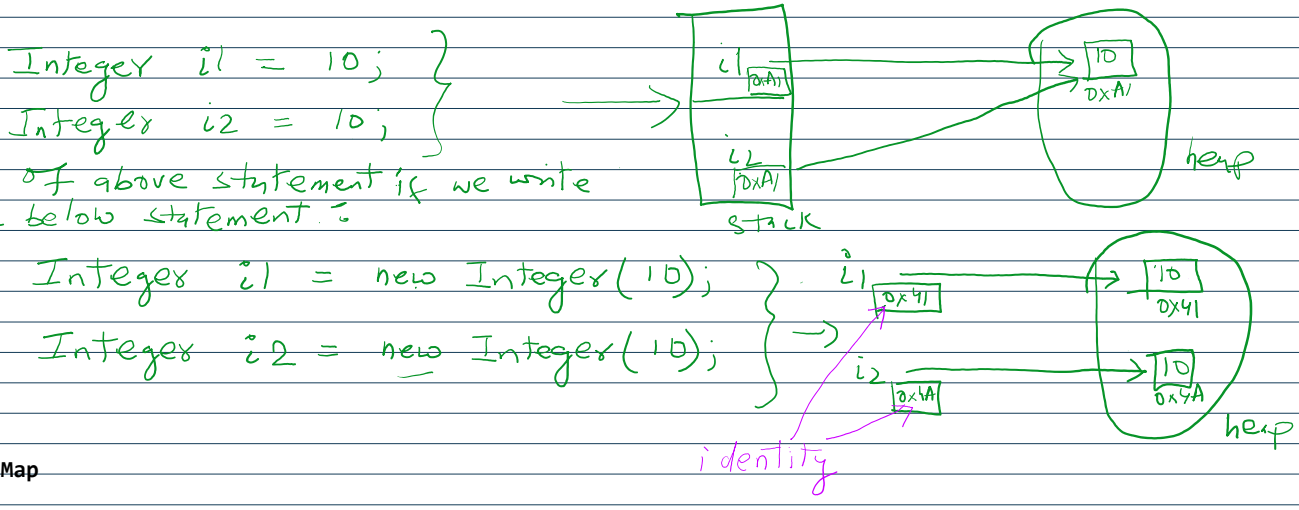
Output:

insertion order is preserved.

```
{Chrinjeevi=1000, Venketesh=800, Nagarjuna=500, Ramesh Babu=900, Rajanikant=1500}
keys: [Chrinjeevi, Venketesh, Nagarjuna, Ramesh Babu, Rajanikant]
Values: [1000, 800, 500, 900, 1500]
Entery Set: [Chrinjeevi=1000, Venketesh=800, Nagarjuna=500, Ramesh Babu=900, Rajanikant=1500]
Chrinjeevi===>1000
Venketesh===>800
Nagarjuna===>500
Ramesh Babu===>900
Rajanikant===>1500
{Chrinjeevi=1000, Venketesh=800, Nagarjuna=10000, Ramesh Babu=900, Rajanikant=1500}
```

LinkedHashMap and LinkedHashSet are commonly used for cache based application.

Link to read more on LinkedHashMap:'
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/LinkedHashMap.html



IdentityHashMap

```
import java.util.HashMap;
public class IdentityHashMapDemo {
    public static void main(String[] args) {
        HashMap m = new HashMap<>();

        Integer i1 = Integer.valueOf(10);
        Integer i2 = Integer.valueOf(10);
        m.put(i1, "James");    // null
        m.put(i2, "Gosling"); // Gosling.
        System.out.println(m);
    }
}
```

10 = ~~James~~ Gosling

@ hexadecimal value

identity is always use by JVM to identify the object

```
        m.put(i2, "Gosling"); // Gosling.
        System.out.println(m);
    }
}
```

Output:

```
null
James
{10=Gosling}
```

What do you mean by identity?

```
import java.util.HashMap;
import java.util.IdentityHashMap;
public class IdentityHashMapDemo {
    public static void main(String[] args) {
        // HashMap m = new HashMap<>();
        IdentityHashMap m = new IdentityHashMap<>();
        Integer i1 = new Integer(10);
        Integer i2 = new Integer(10);
        System.out.println(m.put(i1, "James")); //null
        System.out.println(m.put(i2, "Gosling"));//null
        System.out.println(m);     //{10=James, 10=Gosling}
    }
}
```

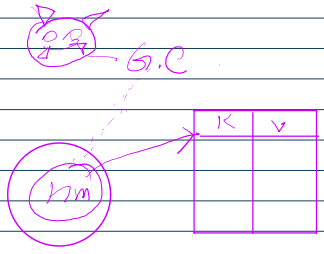
Output:

```
null
null
{10=James, 10=Gosling}
```

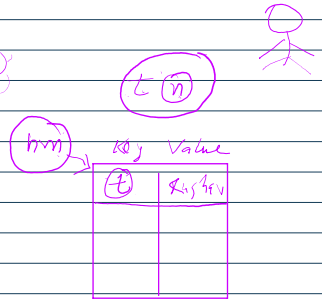
In case of HashMap JVM will use .equals() method to identify, duplicate keys which is made for content comparison.
But in case of IdentityHashMap JVM will use "==" operator to identify, duplicate keys, which is meant for reference comparison (address comparison)

Link to study more about IdentityHashMap
https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/IdentityHashMap.html

WeakHashMap




```
HashMap m = new HashMap();
Temp t = new Temp();
m.put(t, "Raghav");
t = null;
System.gc();
Thread.sleep(5000);
System.out.println(m);
```




```
finalize() {
    clean / update
}
```
This function is called

This function is called by GC before clean up activities

```
public class Temp {
    @Override
    public String toString() {
        return "Temp";
    }
    @Override
    public void finalize(){
        System.out.println("Finalize method called.");
    }
}
```

```
import java.util.HashMap;
public class WeakHashMapDemo {
    public static void main(String[] args) throws InterruptedException {
        HashMap m = new HashMap<>();
        Temp t = new Temp();
        m.put(t, "Raghav");
        t = null;
        System.gc();
        Thread.sleep(3000);
        System.out.println(m);
    }
}
```

stronger than gc()

Output:

```
{Temp=Raghav}
```

```
import java.util.HashMap;
import java.util.WeakHashMap;
public class WeakHashMapDemo {
    public static void main(String[] args) throws InterruptedException {
        // HashMap m = new HashMap<>();
        WeakHashMap m = new WeakHashMap<>();
        Temp t = new Temp();
        m.put(t, "Raghav");
        t = null;
        System.gc();
        Thread.sleep(3000);
        System.out.println(m);
    }
}
```

here gc() has successfully called the finalize() method.
here t is deleted by the gc(), because Map object is weak.

Output:

```
Finalize method called.
{}
```

WeekHashMap is exactly same as HashMap, except the following differences:
In case of Map, even though object does not have any reference, it is, eligible for GC. If it is associated with HashMap, that is HashMap dominates garbage collector. It means garbage collector will not be able to delete that object.
But in case of weak hash map. if the object does not contain any reference, it is eligible for GC. Even though object associated with weakHashMap garbage collector dominates the WeakHashMap. Hence, the entry is deleted.

SortedMap

14 May 2025 10:09

It is the child interface of Map interface. If we want to represent a group of key - to some sorting order of keys. then we should go for SortedMap interface. Sorting is based on key, but not based on value.

Map(I)
|
SortedMap(I)

Object firstKey() → 101

Object lastKey() → 121

SortedMap headMap(112) → {101=A, 105=B, 109=C, 110=D}

SortedMap tailMap(112) → {112=E, 113=F, 115=G, 121=H}

SortedMap subMap(109, 115) → {109=C, 110=D, 112=E, 113=F}

Comparator comparator()

K	V
101	A
105	B
109	C
110	D
112	E
113	F
115	G
121	H

SortedMap (I)
|
NavigableMap(I)
|
TreeMap.

Red-Black Tree properties

1. Every node is either red or black.
2. The root is always black.
3. All leaves (null pointers) are considered black.
4. Red nodes cannot have red children.
5. Every path from a node to its decendent leaves contains the same number of black nodes. This is called black-height of the tree.

Why use Red-Black Trees?

⇒ To maintain balance while performing inserts and deletes

⇒ To guarantee logarithmic height

⇒ Used in many real-world aplication like Java's TreeMap, C++ STL's map and set. Linux process scheduler, etc.

public interface SortedMap<K,V>
extends SequencedMap<K,V>

A Map that further provides a total ordering on its keys. The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the entrySet, keySet and values methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of SortedSet.)

All keys inserted into a sorted map must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such keys must be mutually comparable: k1.compareTo(k2) (or comparator.compare(k1, k2)) must not throw a ClassCastException for any keys k1 and k2 in the sorted map. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a ClassCastException.

Note that the ordering maintained by a sorted map (whether or not an explicit comparator is provided) must be consistent with equals if the sorted map is to correctly implement the Map interface. (See the Comparable interface or Comparator interface for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a tree map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

All general-purpose sorted map implementation classes should provide four "standard" constructors. It is not possible to enforce this recommendation though as required constructors cannot be specified by interfaces. The expected "standard" constructors for all sorted map implementations are:

1. A void (no arguments) constructor, which creates an empty sorted map sorted according to the natural ordering of its keys.
2. A constructor with a single argument of type Comparator, which creates an empty sorted map sorted according to the specified comparator.
3. A constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument, sorted according to the keys' natural ordering.
4. A constructor with a single argument of type SortedMap, which creates a new sorted map with the same key-value mappings and the same ordering as the input sorted map.

Note: several methods return submaps with restricted key ranges. Such ranges are half-open, that is, they include their low endpoint but not their high endpoint (where applicable). If you need a closed range (which includes both endpoints), and the key type allows for calculation of the successor of a given key, merely request the subrange from lowEndpoint to successor(highEndpoint). For example, suppose that m is a map whose keys are strings. The following idiom obtains a view containing all of the key-value mappings in m whose keys are between low and high, inclusive:

```
SortedMap<String, V> sub = m.subMap(low, high+"\0");
```

A similar technique can be used to generate an open range (which contains neither endpoint). The following idiom obtains a view containing all of the key-value mappings in m whose keys are between low and high, exclusive:

```
SortedMap<String, V> sub = m.subMap(low+"\0", high);
```

This interface is a member of the Java Collections Framework.

Since:
1.2

TreeMap

- **The underlying data structure is red-black tree.**

Concept of RED-BLACK Tree:

What is a Red-Black Tree?

A Red-Black Tree is a self-balancing binary search tree where each node has an additional attribute: a color, which can be either red or black. The primary objective of these trees is to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation.
Properties of Red-Black Trees

A Red-Black Tree have the following properties:

Node Color: Each node is either red or black.
Root Property: The root of the tree is always black.
Red Property: Red nodes cannot have red children (no two consecutive red nodes on any path).
Black Property: Every path from a node to its descendant null nodes (leaves) has the same number of black nodes.
Leaf Property: All leaves (NIL nodes) are black.

These properties ensure that the longest path from the root to any leaf is no more than twice as long as the shortest path, maintaining the tree’s balance and efficient performance.

- Insertion order is not preserved and it is based on some sorting order of keys.
- Duplicate keys are not allowed, but values can be duplicated.
- If we are depending on default natural sorting order then. keys should be homogeneous and comparable. Otherwise we will get runtime exception, saying, ClassCastException.
  If we are defining our own sorting by comparator, then keys needed not to be homogeneous and Comparable. We can take heterogeneous, non-comparable objects also.
- Whether we are depending on default natural sorting order or customized sorting order, there are no restriction for values. We can store heterogeneous, non-comparable objects also.
- Null is not accepted at all in keys.

Constructors of TreeMap
1. TreeMap t = new TreeMap(); //DNSO of key
2. TreeMap t = new TreeMap(Comparator c); //for customized sorting order
3. TreeMap t = new TreeMap(Map m);
4. TreeMap t = new TreeMap(SortedMap m);

Official Documentation:

public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

Note that the ordering maintained by a tree map, like any sorted map, and whether or not an explicit comparator is provided, must be consistent with equals if this sorted map is to correctly implement the Map interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Map interface is defined in terms of the equals operation, but a sorted map performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the sorted map, equal. The behavior of a sorted map is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Map interface.

Note that this implementation is not synchronized. If multiple threads access a map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with an existing key is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedSortedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
```

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are fail-fast: if the map is structurally 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.

The methods ceilingEntry(K), firstEntry(), floorEntry(K), higherEntry(K), lastEntry(), lowerEntry(K), pollFirstEntry(), and pollLastEntry() return Map.Entry instances that represent snapshots of mappings as of the time of the call. They do not support mutation of the underlying map via the optional setValue method.

The putFirst and putLast methods of this class throw UnsupportedOperationException. The encounter order of mappings is determined by the comparison method; therefore, explicit positioning is not supported.

This class is a member of the Java Collections Framework.

Since:
1.2

Constructors

Constructor	Description
`TreeMap()`	Constructs a new, empty tree map, using the natural ordering of its keys.
`TreeMap(Comparator<? super K> comparator)`	Constructs a new, empty tree map, ordered according to the given comparator.
`TreeMap(Map<? extends K, ? extends V> m)`	Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
`TreeMap(SortedMap<K, ? extends V> m)`	Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

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

Modifier and Type	Method	Description
`Map.Entry<K,V>`	`ceilingEntry(K key)`	Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
`K`	`ceilingKey(K key)`	Returns the least key greater than or equal to the given key, or `null` if there is no such key.
`void`	`clear()`	Removes all of the mappings from this map.
`Object`	`clone()`	Returns a shallow copy of this `TreeMap` instance.
`Comparator<? super K>`	`comparator()`	Returns the comparator used to order the keys in this map, or `null` if this map uses the natural ordering of its keys.
`V`	`compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`	Attempts to compute a mapping for the specified key and its current mapped value, or `null` if there is no current mapping (optional operation).
`V`	`computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)`	If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null` (optional operation).
`V`	`computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)`	If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value (optional operation).
`boolean`	`containsKey(Object key)`	Returns `true` if this map contains a mapping for the specified key.

boolean	containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
NavigableSet<K>	descendingKeySet()	Returns a reverse order NavigableSet view of the keys contained in this map.
NavigableMap<K,V>	descendingMap()	Returns a reverse order view of the mappings contained in this map.
Set<Map.Entry<K,V>>	entrySet()	Returns a Set view of the mappings contained in this map.
Map.Entry<K,V>	firstEntry()	Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
K	firstKey()	Returns the first (lowest) key currently in this map.
Map.Entry<K,V>	floorEntry(K key)	Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
K	floorKey(K key)	Returns the greatest key less than or equal to the given key, or null if there is no such key.
V	get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
SortedMap<K,V>	headMap(K toKey)	Returns a view of the portion of this map whose keys are strictly less than toKey.
NavigableMap<K,V>	headMap(K toKey, boolean inclusive)	Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.

Map.Entry<K,V>	higherEntry(K key)	Returns a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.
K	higherKey(K key)	Returns the least key strictly greater than the given key, or null if there is no such key.
Set<K>	keySet()	Returns a Set view of the keys contained in this map.
Map.Entry<K,V>	lastEntry()	Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
K	lastKey()	Returns the last (highest) key currently in this map.
Map.Entry<K,V>	lowerEntry(K key)	Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K	lowerKey(K key)	Returns the greatest key strictly less than the given key, or null if there is no such key.
V	merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value (optional operation).
NavigableSet<K>	navigableKeySet()	Returns a NavigableSet view of the keys contained in this map.
Map.Entry<K,V>	pollFirstEntry()	Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty (optional operation).

Map.Entry<K,V>	pollLastEntry()	Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty (optional operation).
V	put(K key, V value)	Associates the specified value with the specified key in this map.
void	putAll(Map<? extends K, ? extends V> map)	Copies all of the mappings from the specified map to this map.
V	putFirst(K k, V v)	Throws UnsupportedOperationException.
V	putLast(K k, V v)	Throws UnsupportedOperationException.
V	remove(Object key)	Removes the mapping for this key from this TreeMap if present.
int	size()	Returns the number of key-value mappings in this map.
NavigableMap<K,V>	subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	Returns a view of the portion of this map whose keys range from fromKey to toKey.
SortedMap<K,V>	subMap(K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V>	tailMap(K fromKey)	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
NavigableMap<K,V>	tailMap(K fromKey, boolean inclusive)	Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.
Collection<V>	values()	Returns a Collection view of the values contained in this map.

Code:

CollectionFramework > Map > TreeMapDemo.java > TreeMapDemo > main(String[])

```java
import java.util.TreeMap;

public class TreeMapDemo {
    Run | Debug
    public static void main(String[] args) {
        TreeMap m = new TreeMap();

        m.put(key:100, value:"Aman");
        m.put(key:87, value:"Naman");
        m.put(key:112, value:"Aditya");
        m.put(key:95, value:"Raghav");
        m.put(key:200, value:"Naveen");
        m.put(key:98, value:null);
        m.put(key:null, value:null);

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

Output:

```
Exception in thread "main" java.lang.NullPointerException
        at java.base/java.util.Objects.requireNonNull(Objects.java:233)
        at java.base/java.util.TreeMap.put(TreeMap.java:809)
        at java.base/java.util.TreeMap.put(TreeMap.java:534)
        at TreeMapDemo.main(TreeMapDemo.java:13)
```

```java
import java.util.TreeMap;
public class TreeMapDemo {
    public static void main(String[] args) {
        TreeMap m = new TreeMap();
        m.put(100, "Aman");
        m.put(87, "Naman");
        m.put(112, "Aditya");
        m.put(95, "Raghav");
        m.put(200, "Naveen");
        m.put(98, null);
        // m.put(null, null);
        System.out.println(m);
    }
}
```

Output:

{87=Naman, 95=Raghav, 98=null, 100=Aman, 112=Aditya, 200=Naveen} – Sorted with keys

```java
import java.util.TreeMap;

public class TreeMapDemo {
    Run | Debug
    public static void main(String[] args) {
        TreeMap m = new TreeMap();

        m.put(key:100, value:"Aman");
        m.put(key:87, value:"Naman");
        m.put(key:112, value:"Aditya");
        m.put(key:95, value:"Raghav");
        m.put(key:200, value:"Naveen");
        m.put(key:98, value:null);
        // m.put(null, null);

        System.out.println(m);

        m.put(key:"Varun", value:"Anuj");

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

Throwable

Exception — Error

RuntimeException — ClassCastException

Hetrogenous object in key is not allowed.

RuntimeException

Output:

```
    }
}
```

Heterogeneous objects are not allowed → RuntimeException

Output:

```
{87=Naman, 95=Raghav, 98=null, 100=Aman, 112=Aditya, 200=Naveen}
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class
java.lang.String (java.lang.Integer 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 TreeMapDemo.main(TreeMapDemo.java:17)
```

```
import java.util.Comparator;
public class MyCompartor implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        return o2.toString().compareTo(o1.toString());
    }
}
```

```
import java.util.TreeMap;
public class TreeMapComparatorDemo {
    public static void main(String[] args) {
        TreeMap t = new TreeMap<>(new MyCompartor());
        t.put("Aditya", "Anchit");
        t.put("Aman", "Naman");
        t.put("Bhavvya", "Riddhi");
        t.put(125, "Surya");
        t.put("Shiv", "Students");
        System.out.println(t);
    }
}
```

Output:

```
{Shiv=Students, Bhavvya=Riddhi, Aman=Naman, Aditya=Anchit, 125=Surya}
```

← Sorted in descending order of 'keys'.

Hashtable

- The underlying data structure for hash table is hash table.
- Insertion order is not preserved and it is based on hash code of keys.
- Duplicate keys are not allowed and values can be duplicated.
- Heterogeneous objects are allowed for both keys and values.
- Null is not allowed for both key and values. Otherwise we will get runtime exception, saying, "NullPointerException".
- It implements Map<k><v>, Serializable and Cloneable interfaces, but not RandomAccess interface.
- Every method present in Hashtable is synchronized and hence Hashtable object is thread safe.
- Hash table is the best choice if our frequent operation is retrieval or search operation.

Constructors:

1. Hashtable h = new Hashtable(); ==> Creates an empty hash table object with default initial capacity 11 and default fill ratio, 0.75.

2. Hashtable h = new Hashtable(int initialCapacity, float fillRatio);

3. Hashtable h = new Hashtable(int initialCapacity);

4. Hashtable h = new Hashtable(Map m); ==>Interconversion between Map object.

Official Documentation:

public class Hashtable<K,V>
extends Dictionary<K,V>

implements Map<K,V>, Cloneable, Serializable
This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

An instance of Hashtable has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. Note that the hash table is open: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. The initial capacity and load factor parameters are merely hints to the implementation. The exact details as to when and whether the rehash method is invoked are implementation-dependent.

Generally, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most Hashtable operations, including get and put).

The initial capacity controls a tradeoff between wasted space and the need for rehash operations, which are time-consuming. No rehash operations will ever occur if the initial capacity is greater than the maximum number of entries the Hashtable will contain divided by its load factor. However, setting the initial capacity too high can waste space.

If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

This example creates a hashtable of numbers. It uses the names of the numbers as keys:

```
   Hashtable<String, Integer> numbers
     = new Hashtable<String, Integer>();
   numbers.put("one", 1);
   numbers.put("two", 2);
   numbers.put("three", 3);
```

To retrieve a number, use the following code:

```
   Integer n = numbers.get("two");
   if (n != null) {
     System.out.println("two = " + n);
   }
```

The iterators returned by the iterator method of the collections returned by all of this class's "collection view methods" are fail-fast: if the Hashtable is structurally 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. The Enumerations returned by Hashtable's keys and elements methods are not fail-fast; if the Hashtable is structurally modified at any time after the enumeration is created then the results of enumerating are undefined.

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 Map interface, making it a member of the Java Collections Framework. Unlike the new collection implementations, Hashtable is synchronized. If a thread-safe implementation is not needed, it is recommended to use HashMap in place of Hashtable. If a thread-safe highly-concurrent implementation is desired, then it is recommended to use ConcurrentHashMap in place of Hashtable.

Since:
    1.0

Code:

```java
import java.util.Hashtable;
public class HashTableDemo {
    public static void main(String[] args) {
        Hashtable<Key, String> h = new Hashtable<Key, String>(25);
        h.put(new Key(5), "A");
        h.put(new Key(2), "B");
        h.put(new Key(6), "C");
        h.put(new Key(15), "D");
        h.put(new Key(17), "E");
        h.put(new Key(23), "F");
        h.put(new Key(15), "G");
        h.put(new Key(17), "H");
        h.put(new Key(19), null);
        System.out.println(h);
    }
}
```

'null' concept is not allowed for 'key' or 'value'

Output:

```
In equals:
In equals:
Exception in thread "main" java.lang.NullPointerException
        at java.base/java.util.Hashtable.put(Hashtable.java:476)
        at HashTableDemo.main(HashTableDemo.java:14)
PS C:\Users\bluej\Documents\Students\RaghavISC\Java4Sem\Coding>
```

```java
import java.util.Hashtable;
public class HashTableDemo {
    public static void main(String[] args) {
        Hashtable<Key, String> h = new Hashtable<Key, String>(25);
        h.put(new Key(5), "A");
        h.put(new Key(2), "B");
        h.put(new Key(6), "C");
        h.put(new Key(15), "D");
        h.put(new Key(17), "E");
        h.put(new Key(23), "F");
        h.put(new Key(15), "G");
        h.put(new Key(17), "H");
        // h.put(new Key(19), null);
        System.out.println(h);
        Key k = new Key(15);
        System.out.println("Value of "+k+" is: "+h.get(k));
    }
}
```

```java
public class Key {
    private int i;
    public Key(int i) {
        this.i = i;
    }
    @Override
    public int hashCode() {
        return i;
    }
    @Override
    public boolean equals(Object obj) {
        Key k = null;
        if(obj instanceof Key){
            k = (Key)obj;
        }
        System.out.println("In equals ");
        if(this.i == k.i){
            return true;
        }
        else{
```

implement any hashing algorithm here.

Overriding of these methods are compulsory to smooth running of Hashtable operations.

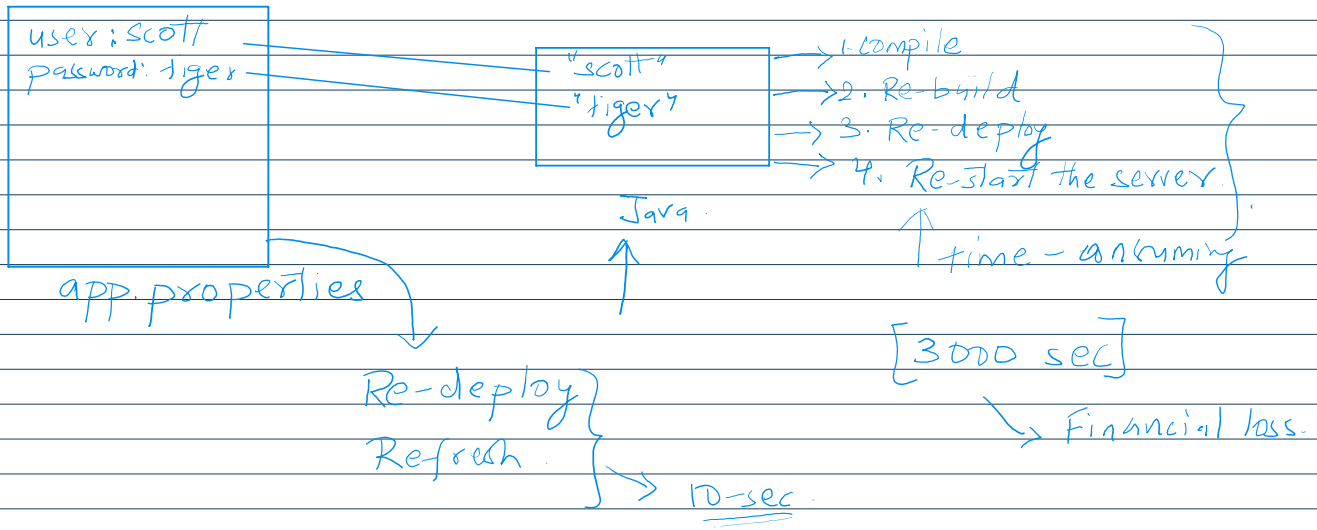
```
            if(this.i == k.i){
                return true;
            }
            else{
                return false;
            }
        }
        @Override
        public String toString() {
            return "Key [i=" + i + "]";
        }

    }
```

Output:

```
In equals
In equals
{Key [i=23]=F, Key [i=17]=H, Key [i=15]=G, Key [i=6]=C, Key [i=5]=A, Key [i=2]=B}
In equals
Value of Key [i=15] is: G
```

Properties

05 June 2025 20:13



In our program, if anything, which changes frequently (like user name, password, mail ids, mobile number, etc.) are not recommended to hard code (Keeping the values directly) in the Java program. Because if there is any change to reflect that change, recompilation, rebuild, redeploy applications are required, even sometimes servers restart also required, which creates business impact to the client.

We can overcome this problem using properties file. Such type of variable thing we have to configure in the properties file that we have to read into Java program. And we can use those properties.
The main advantage of this approach is if there is a change in properties file to reflect that change, just redeployment is enough, which would not create any business impact to the client.

We can use Java **Properties** object to hold properties which are coming from properties file.

```
Properties p = new Properties();
```

In normal map (like HashMap, Hashtable, TreeMap) key-value can be any type. But in case of Properties, key-value should be String type.

Methods:

```
String setProperty(String pname, String pvalue);
```

Old value — If the specified property already available, then old value will be replaced with new values and returns the old value. This function is used to set a new property.

```
String getProperty(String pname);
```

To get the value associated with the specified property. If the specified property is not available, then this method returns null.

```
Enumeration propertyNames();
```

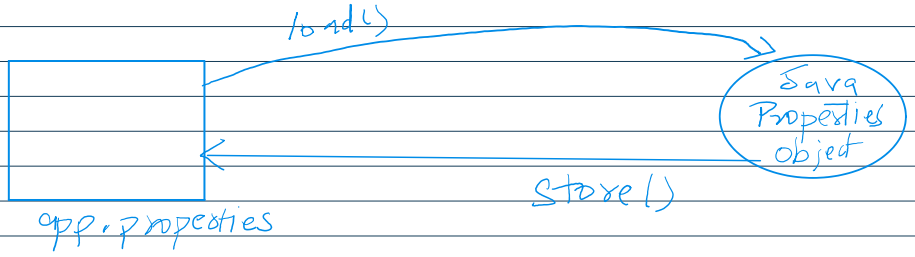
Returns all property name present in properties object.

```
void load(InputStream is);
```

To load properties from properties filed into Java properties object.

```
void store(OutputStream os);
```

To store proper ties from Java. Properties object into properties file.



Official Documentation:

public class Properties
extends Hashtable<Object,Object>

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string.

A property list can contain another property list as its "defaults"; this second property list is searched if the property key is not found in the original property list.

Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not Strings. The setProperty method should be used instead. If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail. Similarly, the call to the propertyNames or list method will fail if it is called on a "compromised" Properties object that contains a non-String key.

The iterators returned by the iterator method of this class's "collection views" (that is, entrySet(), keySet(), and values()) may not fail-fast (unlike the Hashtable implementation). These iterators are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.

The load(Reader) / store(Writer, String) methods load and store properties from and to a character based stream in a simple line-oriented format specified below. The load(InputStream) / store(OutputStream, String) methods work the same way as the load(Reader)/store(Writer, String) pair, except the input/output stream is encoded in ISO 8859-1 character encoding. Characters that cannot be directly represented in this encoding can be written using Unicode escapes as defined in section 3.3 of The Java Language Specification; only a single 'u' character is allowed in an escape sequence.

The loadFromXML(InputStream) and storeToXML(OutputStream, String, String) methods load and store properties in a simple XML format. By default the UTF-8 character encoding is used, however a specific encoding may be specified if required. Implementations are required to support UTF-8 and UTF-16 and may support other encodings. An XML properties document has the following DOCTYPE declaration:

<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

Note that the system URI (http://java.sun.com/dtd/properties.dtd) is not accessed when exporting or importing properties; it merely serves as a string to uniquely identify the DTD, which is:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- DTD for properties -->
```

```
    <!ELEMENT properties ( comment?, entry* ) >

    <!ATTLIST properties version CDATA #FIXED "1.0">

    <!ELEMENT comment (#PCDATA) >

    <!ELEMENT entry (#PCDATA) >

    <!ATTLIST entry key CDATA #REQUIRED>
```

This class is thread-safe: multiple threads can share a single Properties object without the need for external synchronization.

API Note:
The Properties class does not inherit the concept of a load factor from its superclass, Hashtable.

Since:
1.0

Constructor Summary

Constructors

Constructor	Description
`Properties()`	Creates an empty property list with no default values.
`Properties(int initialCapacity)`	Creates an empty property list with no default values, and with an initial size accommodating the specified number of elements without the need to dynamically resize.
`Properties(Properties defaults)`	Creates an empty property list with the specified defaults.

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

Modifier and Type	Method	Description
`String`	`getProperty(String key)`	Searches for the property with the specified key in this property list.
`String`	`getProperty(String key, String defaultValue)`	Searches for the property with the specified key in this property list.
`void`	`list(PrintStream out)`	Prints this property list out to the specified output stream.
`void`	`list(PrintWriter out)`	Prints this property list out to the specified output stream.
`void`	`load(InputStream inStream)`	Reads a property list (key and element pairs) from the input byte stream.
`void`	`load(Reader reader)`	Reads a property list (key and element pairs) from the input character stream in a simple line-oriented format.
`void`	`loadFromXML(InputStream in)`	Loads all of the properties represented by the XML document on the specified input stream into this properties table.
`Enumeration<?>`	`propertyNames()`	Returns an enumeration of all the keys in this property list, including distinct keys in the default property list if a key of the same name has not already been found from the main properties list.
`void`	`save(OutputStream out, String comments)`	**Deprecated.** This method does not throw an IOException if an I/O error occurs while saving the property list.
`Object`	`setProperty(String key, String value)`	Calls the `Hashtable` method `put`.

void	store(OutputStream out, String comments)	Writes this property list (key and element pairs) in this Properties table to the output stream in a format suitable for loading into a Properties table using the load(InputStream) method.
void	store(Writer writer, String comments)	Writes this property list (key and element pairs) in this Properties table to the output character stream in a format suitable for using the load(Reader) method.
void	storeToXML(OutputStream os, String comment)	Emits an XML document representing all of the properties contained in this table.
void	storeToXML(OutputStream os, String comment, String encoding)	Emits an XML document representing all of the properties contained in this table, using the specified encoding.
void	storeToXML(OutputStream os, String comment, Charset charset)	Emits an XML document representing all of the properties contained in this table, using the specified encoding.
Set<String>	stringPropertyNames()	Returns an unmodifiable set of keys from this property list where the key and its corresponding value are strings, including distinct keys in the default property list if a key of the same name has not already been found from the main properties list.

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;
public class PropertiesDemo {
    public static void main(String[] args) {
        Properties p = new Properties();
        try {
            p.load(new FileInputStream("./CollectionFramework/Map/app.properties"));
            String str = p.getProperty("user");
            System.out.println("value: " + str);
            p.setProperty("newuser2", "shivin");
            p.setProperty("newpassword2", "12345@09876abc");
            p.store(new FileOutputStream("./CollectionFramework/Map/app.properties"), "Just
updated");
        }
        catch(FileNotFoundException e){
            System.out.println("File nahi mila!!!!"+"\n"+e.getMessage());
        }
        catch (IOException e) {
            // TODO Auto-generated catch block
            System.out.println("Network Error"+"\n"+e.getMessage());
        }
    }
}
```

Output:

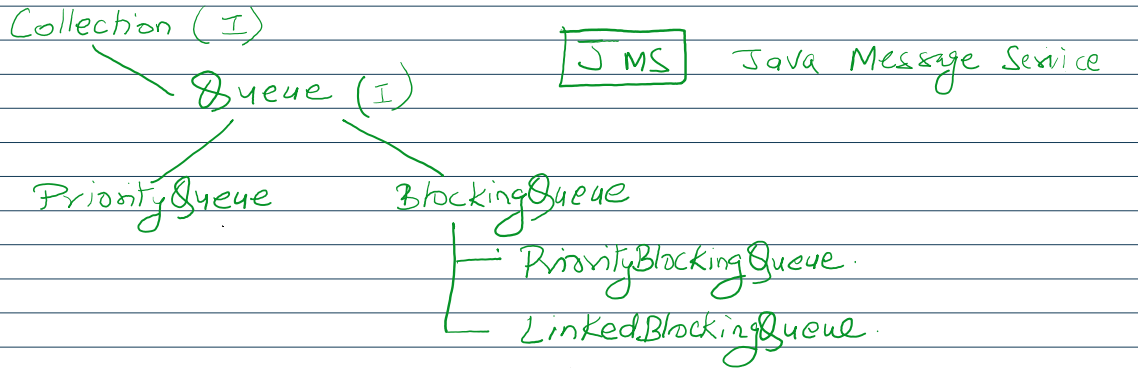
```
value: raghav
```

CollectionFramework > Map > app.properties

```
1  #Just updated
2  #Thu Jun 05 20:55:17 IST 2025
3  newpassword1=12345@09876
4  newpassword2=12345@09876abc
5  newuser1=shiv
6  newuser2=shivin
7  password=12345@09876
8  user=raghav
9
```

Enhancement in Collections

10 June 2025 18:38



Queue interface

- It is a child interface of Collection.
- If we want to represent a group of individual objects "Prior to process". Then we should go for Queue interface. For example before sending SMS message all mobile numbers we have to store in some data structure. In which order we add mobile number in the same order only message should be delivered. For this FIFO requirement, Queue is the best choice.
- Queue follows FIFO, but based on our requirement we can implement our own priority order also. (Priority queue).
- From. 1.5 version onwards. LinkedList class also implements Queue interface. LinkedList based implementation of Queue always follow FIFO.

Queue (I) Specific methods :-

1. boolean offer (Object o) ← to add object into the queue.
2. Object poll () ← remove and return head element of queue.
3. Object peek () ← it returns the head element of queue. if queue is empty then this method returns null.
4. Object element() ← it returns the head element of queue. If queue is empty then this method raises runtime exception i.e; "NoSuchElementException"
5. Object remove () ← it returns the head element of the queue. If queue is empty then this method raises runtime exception i.e "NoSuchElementException"

Official Documentation:

public interface Queue<E>
extends Collection<E>
A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.

Summary of Queue methods

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the head of the queue is that element which would be removed by a call to remove() or poll(). In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

The offer method inserts an element if possible, otherwise returning false. This differs from the Collection.add method, which can fail to add an element only by throwing an unchecked exception. The offer method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.

The remove() and poll() methods remove and return the head of the queue. Exactly which element is removed from the queue is a function of the queue's ordering policy, which differs from implementation to implementation. The remove() and poll() methods differ only in their behavior when the queue is empty: the remove() method throws an exception, while the poll() method returns null.

The element() and peek() methods return, but do not remove, the head of the queue.

The Queue interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the BlockingQueue interface, which extends this interface.

Queue implementations generally do not allow insertion of null elements, although some implementations, such as LinkedList, do not prohibit insertion of null. Even in the implementations that permit it, null should not be inserted into a Queue, as null is also used as a special return value by the poll method to indicate that the queue contains no elements.

Queue implementations generally do not define element-based versions of methods equals and hashCode but instead inherit the identity based versions from class Object, because element-based equality is not always well-defined for queues with the same elements but different ordering properties.

This interface is a member of the Java Collections Framework.

Since:
1.5

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
boolean	add(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
E	element()	Retrieves, but does not remove, the head of this queue.
boolean	offer(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E	peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.
E	remove()	Retrieves and removes the head of this queue.

Priority Queue

- **If we want to represent a group of individual objects prior to processing according to some**

priority then we should go for priority queue.

- **The priority can be either *default natural sorting order* Or *customized sorting order* defined by Comparator.**
- **Insertion order is not preserved and it is based on some priority.**
- **If we are depending on default natural sorting order, it is compulsory object should be homogeneous and Comparable otherwise we will get runtime exception saying *"ClassCastException"*.**
- **If we are defining our own sorting by Comparator, then objects need not be homogeneous and Comparable.**
- **Null is not allowed at all.**

Constructors

PriorityQueue pq = new PriorityQueue(); ==> It creates an empty priority queue with default initial capacity 11 and all objects is inserted according to default natural sorting order.

PriortyQueue pq = new PriorityQueue(int initialCapacity);

PriortyQueue pq = new PriorityQueue(int initialCapacity, Comparator c);

PriortyQueue pq = new PriorityQueue(Comparator c);

PriortyQueue pq = new PriorityQueue(SortedSet s);

PriortyQueue pq = new PriorityQueue(Collection c); } ← interconversion constructor.

Official Documentation:

public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).

The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.

A priority queue is unbounded, but has an internal capacity governing the size of an array used to store the elements on the queue. It is always at least as large as the queue size. As elements are added to a priority queue, its capacity grows automatically. The details of the growth policy are not specified.

This class and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The Iterator provided in method iterator() and the Spliterator provided in method spliterator() are not guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray()).

Note that this implementation is not synchronized. Multiple threads should not access a PriorityQueue instance concurrently if any of the threads modifies the queue. Instead, use the thread-safe PriorityBlockingQueue class.

Implementation note: this implementation provides O(log(n)) time for the enqueuing and dequeuing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).

This class is a member of the Java Collections Framework.

Since:
1.5
https://docs.oracle.com/en/java/javase/24/docs/api//java.base/java/util/PriorityQueue.html

To know more about methods consider the above link.

```
import java.util.PriorityQueue;
public class PriorityQueueDemo{
    public static void main(String[] args) {
        PriorityQueue pq = new PriorityQueue();
        System.out.println(pq.peek()); //null
        System.out.println(pq.element()); //NoSuchElementException
        for (int i = 0; i < 11; i++) {
            pq.offer(i);
        }
        System.out.println(pq);
    }
}
```

Output:

```
null
Exception in thread "main" java.util.NoSuchElementException
        at java.base/java.util.AbstractQueue.element(AbstractQueue.java:136)
        at PriorityQueueDemo.main(PriorityQueueDemo.java:7)
```

```
import java.util.PriorityQueue;
import java.util.Random;
import java.util.Scanner;
public class PriorityQueueDemo{
    public static void main(String[] args) {
        PriorityQueue pq = new PriorityQueue();
        System.out.println(pq.peek()); //null
        //System.out.println(pq.element()); //NoSuchElementException
        for (int i = 0; i < 11; i++) {
            int x = new Random().nextInt(10, 100);
            System.out.println(x);
            pq.offer(x);
        }
        System.out.println(pq);
        while(!pq.isEmpty()){
            System.out.println("Press y/n to poll elements from priority queue: ");
            String x = new Scanner(System.in).next().toLowerCase();
            if (x.charAt(0) == 'n'){
                break;
            }
            System.out.println(pq.poll());
        }
    }
}
```

Output:

```
null
33
91
87
48
43
83
14
73
18
73
37
[14, 18, 33, 43, 37, 87, 83, 91, 73, 73, 48]
Press y/n to poll elements from priority queue:
y
14
Press y/n to poll elements from priority queue:
y
18
Press y/n to poll elements from priority queue:
y
33
Press y/n to poll elements from priority queue:
y
37
Press y/n to poll elements from priority queue:
y
43
Press y/n to poll elements from priority queue:
y
48
Press y/n to poll elements from priority queue:
y
73
Press y/n to poll elements from priority queue:
y
73
Press y/n to poll elements from priority queue:
y
83
Press y/n to poll elements from priority queue:
y
87
Press y/n to poll elements from priority queue:
y
91
```

Example with Comparator

```
import java.util.Comparator;
public class Mycomparator implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        return o2.toString().compareTo(o2.toString());
    }
}
```

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

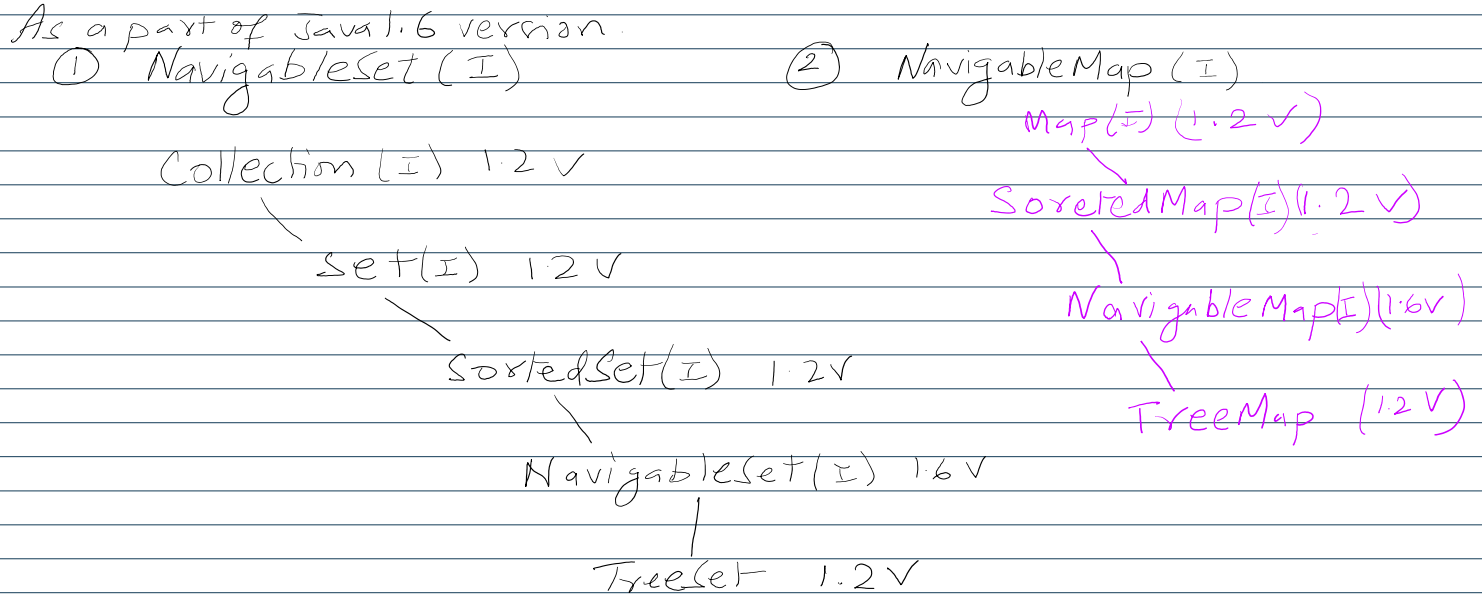
```
        PriorityQueue pq = new PriorityQueue<>(15, new MyCompartor());
        pq.add("Atrijo");
        pq.add("Raghav");
        pq.add("Arjun");
        pq.add("Hitansh");
        pq.add("Aadya");
        pq.add("Kanav");
        System.out.println(pq);
        while(!pq.isEmpty()){
            System.out.println(pq.poll());
        }
    }
}
```

Output:

```
[Raghav, Hitansh, Kanav, Atrijo, Aadya, Arjun]
Raghav
Kanav
Hitansh
Atrijo
Arjun
Aadya
```

Enhancement in Collection 1.6

26 June 2025 18:33



NavigableSet interface: It is a child interface of Sorted Set and it defines several methods for navigation purpose.

NavigableSet defines the following methods:

floor(<E>) ==> It returns highest element which is <= E.
lower(<E>) ==> It returns highest element which is < E.
ceiling(<E>) ==> It returns lowest element which is >= E.
higher(<E>) ==> It returns lowest element which is > E.

pollFirst() ==> Remove and return first element.
pollLast() ==> Remove and return last element.

descendingSet() ==> It returns navigable set in reverse order.

Methods from SortedSet: first(), last(), headSet(), tailSet(), subSet(),…

TreeSet <E> t = new TreeSet<E>();

```
package EnhansementInJavaCollection;
import java.util.TreeSet;
public class NavigableSetDemo {
    public static void main(String[] args) {
        TreeSet<Integer> t = new TreeSet<Integer>();
        t.add(5000);
        t.add(4000);
        t.add(1000);
        t.add(6000);
        t.add(2000);
        t.add(3000);
        System.out.println(t); //[1000, 2000, 3000, 4000, 5000, 6000]
        System.out.println(t.ceiling(2000)); // 2000
        System.out.println(t.higher(2000)); // 3000
        System.out.println(t.floor(3000)); // 3000
        System.out.println(t.lower(3000)); // 2000
        System.out.println(t.pollFirst()); //1000
```

```
        System.out.println(t.pollFirst()); //1000
        System.out.println(t); //[2000, 3000, 4000, 5000, 6000]
        System.out.println(t.pollLast()); //6000
        System.out.println(t);//[2000, 3000, 4000, 5000]
        System.out.println(t.descendingSet()); //[5000, 4000, 3000, 2000]
        System.out.println("Original object: "+ t); //[2000, 3000, 4000, 5000]
    }
}
```

Output:

```
[1000, 2000, 3000, 4000, 5000, 6000]
2000
3000
3000
2000
1000
[2000, 3000, 4000, 5000, 6000]
6000
[2000, 3000, 4000, 5000]
[5000, 4000, 3000, 2000]
Original object: [2000, 3000, 4000, 5000]
```

NavigableMap (I)

NavigableMap (I)
Map (I) (1.2 V)
SortedMap (I) (1.2 V)
NavigableMap (I) (1.6 V)
TreeMap (1.2 V)

Dictionary (Java 1.0)
(key, value) are called Entry.
key = value

Python:
key: value
Dictionary (dict)

It is a child interface of SortedMap (I)
NavigableMap (I) defines the following methods

floorKey (e)
lowerKey (e)
ceilingKey (e)
higherKey ()
pollFirstEntry ()
pollLastEntry ()
descendingMap () → returns NavigableMap object

```
package EnhansementInJavaCollection;
```

```java
import java.util.TreeMap;
public class NavigableMapDemo {
    public static void main(String[] args) {
        TreeMap<String, String> t = new TreeMap<String, String>();
        t.put("C", "Cat");
        t.put("D", "Dog");
        t.put("B", "Boy");
        t.put("A", "Apple");
        t.put("G", "Gun");
        t.put("F", "Fish");
        t.put("E", "Elephant");
        System.out.println(t);
        System.out.println(t.ceilingKey("C")); //C
        System.out.println(t.higherKey("C")); //D
        System.out.println(t.floorKey("E")); //E
        System.out.println(t.lowerKey("E")); //D
        System.out.println(t.pollFirstEntry()); //A=Apple
        System.out.println(t.pollLastEntry()); //G=Gun
        System.out.println(t.descendingMap());//{F=Fish, E=Elephant, D=Dog, C=Cat, B=Boy}
        System.out.println(t); //{B=Boy, C=Cat, D=Dog, E=Elephant, F= Fish}
    }
}
```

Output:

```
{A=Apple, B=Boy, C=Cat, D=Dog, E=Elephant, F=Fish, G=Gun}
C
D
E
D
A=Apple
G=Gun
{F=Fish, E=Elephant, D=Dog, C=Cat, B=Boy}
{B=Boy, C=Cat, D=Dog, E=Elephant, F=Fish}
```

Official Documentation:

public interface NavigableMap<K,V>
extends SortedMap<K,V>

A SortedMap extended with navigation methods returning the closest matches for given search targets. Methods lowerEntry(K), floorEntry(K), ceilingEntry(K), and higherEntry(K) return Map.Entry objects associated with keys respectively less than, less than or equal, greater than or equal, and greater than a given key, returning null if there is no such key. Similarly, methods lowerKey(K), floorKey(K), ceilingKey(K), and higherKey(K) return only the associated keys. All of these methods are designed for locating, not traversing entries.

A NavigableMap may be accessed and traversed in either ascending or descending key order. The descendingMap() method returns a view of the map with the senses of all relational and directional methods inverted. The performance of ascending operations and views is likely to be faster than that of descending ones. Methods subMap(K, boolean, K, boolean), headMap(K, boolean), and tailMap(K, boolean) differ from the like-named SortedMap methods in accepting additional arguments describing whether lower and upper bounds are inclusive versus exclusive. Submaps of any NavigableMap must implement the NavigableMap interface.

This interface additionally defines methods firstEntry(), pollFirstEntry(), lastEntry(), and pollLastEntry() that return and/or remove the least and greatest mappings, if any exist, else returning null.

The methods ceilingEntry(K), firstEntry(), floorEntry(K), higherEntry(K), lastEntry(), lowerEntry(K), pollFirstEntry(), and pollLastEntry() return Map.Entry instances that represent snapshots of mappings as of the time of the call. They do not support mutation of the underlying map via the optional setValue method.

Methods subMap(K, K), headMap(K), and tailMap(K) are specified to return SortedMap to allow existing implementations of SortedMap to be compatibly retrofitted to implement NavigableMap, but extensions and implementations of this interface are encouraged to override these methods to return NavigableMap. Similarly, SortedMap.keySet() can be overridden to return NavigableSet.

This interface is a member of the Java Collections Framework.

Since:
1.6

https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/NavigableMap.html

Click the link to explore more.

Official Documentation of NavigableSet

public interface NavigableSet<E>
extends SortedSet<E>

A SortedSet extended with navigation methods reporting closest matches for given search targets. Methods lower(E), floor(E), ceiling(E), and higher(E) return elements respectively less than, less than or equal, greater than or equal, and greater than a given element, returning null if there is no such element.

A NavigableSet may be accessed and traversed in either ascending or descending order. The descendingSet() method returns a view of the set with the senses of all relational and directional methods inverted. The performance of ascending operations and views is likely to be faster than that of descending ones. This interface additionally defines methods pollFirst() and pollLast() that return and remove the lowest and highest element, if one exists, else returning null. Methods subSet(E, boolean, E, boolean), headSet(E, boolean), and tailSet(E, boolean) differ from the like-named SortedSet methods in accepting additional arguments describing whether lower and upper bounds are inclusive versus exclusive. Subsets of any NavigableSet must implement the NavigableSet interface.

The return values of navigation methods may be ambiguous in implementations that permit null elements. However, even in this case the result can be disambiguated by checking contains(null). To avoid such issues, implementations of this interface are encouraged to not permit insertion of null elements. (Note that sorted sets of Comparable elements intrinsically do not permit null.)

Methods subSet(E, E), headSet(E), and tailSet(E) are specified to return SortedSet to allow existing implementations of SortedSet to be compatibly retrofitted to implement NavigableSet, but extensions and implementations of this interface are encouraged to override these methods to return NavigableSet.

This interface is a member of the Java Collections Framework.

Since:
1.6

https://docs.oracle.com/en/java/javase/24/docs/api//java.base/java/util/NavigableSet.html

Click this link to know more about NavigableSet.

Utility Classes

07 July 2025 18:48

Collections: Collections class defines several utility methods for collections objects like Sorting, searching, reversing, etc.

Sorting elements of List:

i. public static void sort(List l);

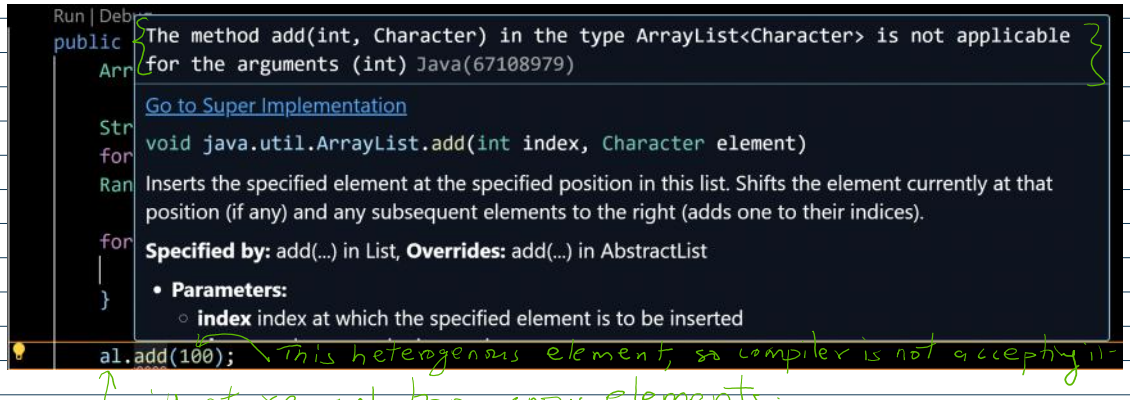
1. D.N.S.O
2. Objects should be homogenous and Comparable otherwise Runtime exception: ClassCastException.
3. If list contain 'null' as object then Runtime exception: NullPointerException.

ii. public static void sort(List l, Comparator c); → C.S.O

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
public class CollectionsSortDemo {
    public static void main(String[] args) {
        ArrayList<Character> al = new ArrayList<Character>();
        String alpha = "";
        for(char i = 'A'; i <= 'Z'; alpha+=i,i++);
        Random r = new Random();
        for(int i = 0; i < 10 + r.nextInt(10); i++){
            al.add(alpha.charAt(r.nextInt(26)));
        }
        System.out.println("Before sorting: " + al);
        Collections.sort(al);
        System.out.println("After sorting: " + al);
    }
}
```

Output:

```
Before sorting: [K, M, B, L, Y, A, C, V, M, B, X, V]
After sorting: [A, B, B, C, K, L, M, M, V, V, X, Y]
```



This heterogenous element, so compiler is not accepting it.

```
index index at which the specified element is to be inserted
al.add(100);
```

This heterogenous element, so compiler is not accepting it.

↑ will store only homogenous elements.

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
public class CollectionsSortDemo {
    public static void main(String[] args) {
        ArrayList<Character> al = new ArrayList<Character>();
        String alpha = "";
        for(char i = 'A'; i <= 'Z'; alpha+=i,i++);
        Random r = new Random();
        for(int i = 0; i < 10 + r.nextInt(10); i++){
            al.add(alpha.charAt(r.nextInt(26)));
        }
        al.add(100);
        System.out.println("Before sorting: " + al);
        Collections.sort(al);
        System.out.println("After sorting: " + al);
    }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The method add(int, Character) in the type ArrayList<Character> is not applicable for the arguments (int)

        at UtilityClasses.CollectionsSortDemo.main(CollectionsSortDemo.java:18)
```

Customized Sorting Order:

```
package UtilityClasses;
import java.util.Comparator;
public class MyComparator implements Comparator{
    @Override
    public int compare(Object o1, Object o2) {
        Character c1 = (Character)o1;
        Character c2 = (Character)o2;
        return c2.compareTo(c1);
    }
}
```

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
public class SortWithComparatorDemo {

    public static void main(String[] args) {
        ArrayList<Character> al = new ArrayList<Character>();
        String alpha = "";
        for(char i = 'A'; i <= 'Z'; alpha+=i,i++);
        Random r = new Random();
        for(int i = 0; i < 10 + r.nextInt(10); i++){
            al.add(alpha.charAt(r.nextInt(26)));
```

```
        }
        System.out.println("Before sorting: " + al);
        Collections.sort(al, new MyComparator());
        System.out.println("After sorting: " + al);
    }
}
```

Output:

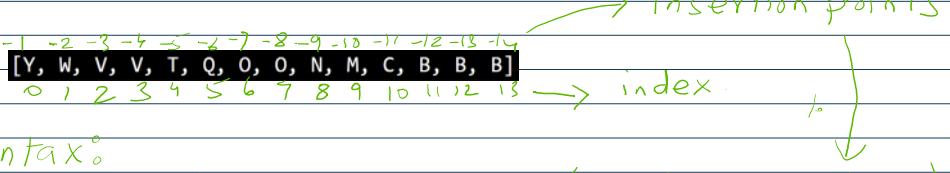
```
Before sorting: [M, Y, V, Q, B, O, V, B, O, C, T, N, W, B]
After sorting: [Y, W, V, V, T, Q, O, O, N, M, C, B, B, B]
```

Searching elements in List

starting element = 0

last element = -1 (↳ (n-1))

n-1-n



① Syntax:

```
public static int binarySearch(List l, Object target);
```

```
Collections.binarySearch(l, "Z"); // -1
Collections.binarySearch(l, "J"); // -11
Collections.binarySearch(l, "N"); // 8
```

② Syntax:-

```
public static int binarySearch(List l, Object target, Comparator c);
```

sorting order of the list is customized by the comparator

Conclusion:-

1. The above search methods will use binary search algorithm.
2. Successful search returns index.
3. Unsuccessful search returns insertion point.
4. Insertion point is the location where we can place the target element in sorted list.
5. Before calling binarySearch() method, compulsory list should be sorted, otherwise we will get unpredictable results.
6. If the list is sorted according to the Comparator then at

sorted, otherwise we will get unpredictable results.

6. If the list is sorted according to the comparator, then at the time of calling binarySearch() method, we have to pass same Comparator object, otherwise we will get unpredictable results.

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
public class CollectionSearchDemo {
    public static void main(String[] args) {
        ArrayList<Character> al = new ArrayList<>();
        String bigAlpha="";
        for (char i = 'A'; i <= 'Z'; i++) {
            bigAlpha +=i;
        }
        Random r = new Random();
        for (int i = 0; i < 10 + r.nextInt(20); i++) {
            al.add(bigAlpha.charAt(r.nextInt(26)));
        }
        System.out.println("List before sorting: " + al);
        Collections.sort(al);
        System.out.println("List after sorting: " + al );
        int in = Collections.binarySearch(al, 'D');
        if(in > -1){
            System.out.println("Element is found at index: "+in);
        }
        else{
            System.out.println("Element insertion point is: "+in);
        }
        in = Collections.binarySearch(al, 'Q');
        if(in > -1){
            System.out.println("Element is found at index: "+in);
        }
        else{
            System.out.println("Element insertion point is: "+in);
        }
        in = Collections.binarySearch(al, 'T');
        if(in > -1){
            System.out.println("Element is found at index: "+in);
        }
        else{
            System.out.println("Element insertion point is: "+in);
        }
    }
}
```

Output:

```
List before sorting: [I, I, V, L, L, O, A, H, F, E, J, P, L, N, A, J, G]
List after sorting: [A, A, E, F, G, H, I, I, J, J, L, L, L, N, O, P, V]
Element insertion point is: -3
Element insertion point is: -17
Element insertion point is: -17
```

```
Element insertion point is: -17
```

Another execution:

```
List before sorting: [P, G, X, A, J, Y, V, E, S, Q, J, B, B, Z, W, C, W, V, L, N, K]
List after sorting: [A, B, B, C, E, G, J, J, K, L, N, P, Q, S, V, V, W, W, X, Y, Z]
Element insertion point is: -5
Element is found at index: 12
Element insertion point is: -15
```

Collections.binarySearch(list, se, comparator)

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
public class CollectionSearchDemoUsingComparator {
    public static void main(String[] args) {
        ArrayList<Character> al = new ArrayList<>();
        String bigAlpha="";
        for (char i = 'A'; i <= 'Z'; i++) {
            bigAlpha +=i;
        }
        Random r = new Random();
        for (int i = 0; i < 10 + r.nextInt(20); i++) {
            al.add(bigAlpha.charAt(r.nextInt(26)));
        }
        System.out.println("List before sorting: " + al);
        MyComparator c = new MyComparator();
        Collections.sort(al, c);
        System.out.println("List after sorting: " + al );
        int in = Collections.binarySearch(al, 'D', c);
        if(in > -1){
            System.out.println("Element is found at index: "+in);
        }
        else{
            System.out.println("Element insertion point is: "+in);
        }
        in = Collections.binarySearch(al, 'Q', c);
        if(in > -1){
            System.out.println("Element is found at index: "+in);
        }
        else{
            System.out.println("Element insertion point is: "+in);
        }
        in = Collections.binarySearch(al, 'T', c);
        if(in > -1){
            System.out.println("Element is found at index: "+in);
        }
        else{
            System.out.println("Element insertion point is: "+in);
        }
    }
}
```

```
package UtilityClasses;
import java.util.Comparator;
public class MyComparator implements Comparator{
    @Override
```

```
    public int compare(Object o1, Object o2) {
        Character c1 = (Character)o1;
        Character c2 = (Character)o2;
        return c2.compareTo(c1);
    }
}
```

Output:

```
List before sorting: [R, N, W, U, E, R, A, Z, E, C, K]
List after sorting: [Z, W, U, R, R, N, K, E, E, C, A]
Element insertion point is: -10
Element insertion point is: -6
Element insertion point is: -4
```

Another execution:

```
List before sorting: [A, U, A, E, B, H, S, B, B, M, D, B, S, U, L, X, F, F, L, U, Q]
List after sorting: [X, U, U, U, S, S, Q, M, L, L, H, F, F, E, D, B, B, B, B, A, A]
Element is found at index: 14
Element is found at index: 6
Element insertion point is: -5
```

Reversing elements in List

```
public static void reverse(List L);
```

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
public class ReversingListDemo {
    public static void main(String[] args) {

        ArrayList<Character> al = new ArrayList<>();
        String bigAlpha="";
        for (char i = 'A'; i <= 'Z'; i++) {
            bigAlpha +=i;
        }
        Random r = new Random();
        for (int i = 0; i < 10 + r.nextInt(20); i++) {
            al.add(bigAlpha.charAt(r.nextInt(26)));
        }
        System.out.println("List before reversing: "+ al);
        Collections.reverse(al);
        System.out.println("List after reverse: " + al);
    }
}
```

Output:

```
List before reversing: [I, L, R, D, Q, Q, S, E, G, C, J, V]
List after reverse: [V, J, C, G, E, S, Q, Q, D, R, L, I]
```

Another implementation:

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Random;
public class ReversingListDemo {
    public static void main(String[] args) {
```

```
        ArrayList<Character> al = new ArrayList<>();
        String bigAlpha="";
        for (char i = 'A'; i <= 'Z'; i++) {
            bigAlpha +=i;
        }
        Random r = new Random();
        for (int i = 0; i < 10 + r.nextInt(20); i++) {
            al.add(bigAlpha.charAt(r.nextInt(26)));
        }
        System.out.println("List before reversing: "+ al);
        Collections.reverse(al);
        System.out.println("List after reverse: " + al);
        Collections.sort(al);
        System.out.println("Default natural sorting order: " + al);
        Collections.reverse(al);
        System.out.println("Descending order: " + al);
    }
}
```

Output:

```
List before reversing: [F, D, S, Y, R, O, Z, O, N, G, P, V, R, Y, J]
List after reverse: [J, Y, R, V, P, G, N, O, Z, O, R, Y, S, D, F]
Default natural sorting order: [D, F, G, J, N, O, O, P, R, R, S, V, Y, Y, Z]
Descending order: [Z, Y, Y, V, S, R, R, P, O, O, N, J, G, F, D]
```

reverseOrder() method is used to get the reversed comparator object.

```
Comparator c1 = Collections.reverseOrder(Comparator c);
```

ascending order ↑

descending order ↑

```
package UtilityClasses;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Random;
public class CollectionsReverseOderUsingComparator {
    public static void main(String[] args) {

        ArrayList<Character> al = new ArrayList<>();
        String bigAlpha="";
        for (char i = 'A'; i <= 'Z'; i++) {
            bigAlpha +=i;
        }
        Random r = new Random();
        for (int i = 0; i < 10 + r.nextInt(20); i++) {
            al.add(bigAlpha.charAt(r.nextInt(26)));
        }
        System.out.println("List before sorting: "+al);
        Collections.sort(al, new MyComparator());
        System.out.println("List in descending order: "+al);
        Comparator c = Collections.reverseOrder( new MyComparator());
        Collections.sort(al, c);
        System.out.println("List in ascending order: "+al);
    }
}
```

Output:

```
List before sorting: [O, B, R, U, X, Y, P, U, H, B, G, D]
List in descending order: [Y, X, U, U, R, P, O, H, G, D, B, B]
List in ascending order: [B, B, D, G, H, O, P, R, U, U, X, Y]
```

Arrays the Utility Class

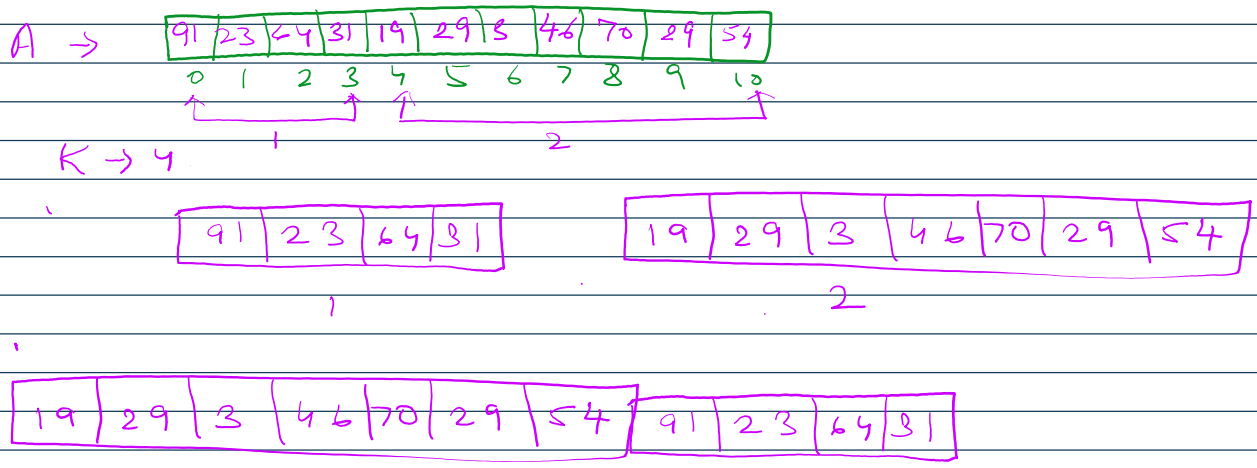
Arrays class is an utility class to define several utility methods for arrays or array.

1. Sorting elements of Array: Arrays class defines the following sort methods to sort elements of primitive type and Object type.
   a. public static void sort(primitive_data_type[] arr); to sort according to DNSO.
   b. public static void sort(Object[] arr); to sort according to CSO.
   c. public static void sort(primitive_data_type[] arr, Comparator c); to sort according to CSO.

Interview Questions

24 February 2025 18:37

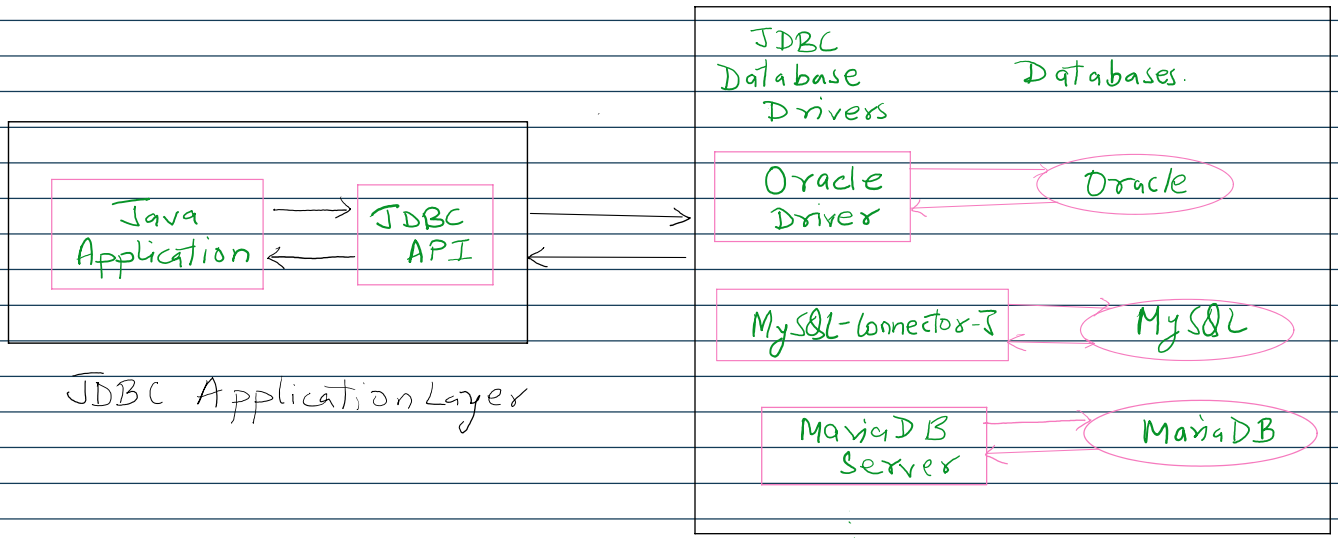
Rotate array by K elements - Block Swap Algorithm



Introduction JDBC

15 April 2025 17:20

JDBC is a standard API for connecting and executing queries with databases. It is part of Java Standard Edition platform and is widely used in enterprise application.



JDBC Architecture

JDBC Architecture Overview

The JDBC API consist of two layers:

- Application Layer: Java application using JDBC API
- JDBC Driver Layer: Interfaces with specific databases(MySQL, Oracle, etc.)

There are 4 types of JDBC drivers:

Type-1: JDBC-ODBC bridge
Type-2: Native-API driver
Type-3: Network protocol driver
Type-4: Thin driver(mostly used driver)

MySQL Connector-J / Oracle

JDBC API

Important Classes and Interfaces

Type 5 driver : 3rd party driver

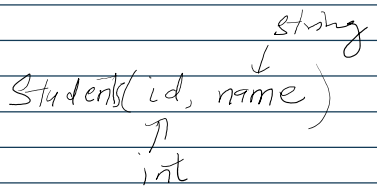
java.sql.DriverManager
java.sql.Connection(I)
java.sql.Statement(I)
java.sql.PreparedStatement(I)
java.sql.CallableStatement(I)
java.sql.ResultSet(I)
java.sql.ResultSetMetaData(I)
java.sql.DatabaseMetaData(I)
java.sql.SQLException

JDBC Workflow

1. **Import JDBC packages**

```
import java.sql.*;
```

```
2. Load and Register the Driver
   Class.forName("com.mysql.cj.jdbc.Driver"); //MySQL

   Class.forName("oracle.jdbc.OracleDriver"); //Oracle

3. Establish a Database connection
   Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/database_name", "root",
   "password");

                Or
   Connection con = DriverManager.getConnection(url, user, password);

4. Create a Statement
   Statement stmt = con.createStatement();

5. Execute Queries
   ResultSet rs = stmt.executeQuery("SELECT * FROM students");

6. Process Results
      while(rs.next()){
          System.out.print("\nID:" + rs.getInt("id")+", Name: " + rs.getString("name"));
      }

7. Close the connection
   rs.close();
   stmt.close();
   con.close();
```



Using PreparedStatement(Parameterized Queries)

Helps prevent SQL injection and optimize performance.

```
String query = "SELECT * FROM students WHERE id = ?";
PreparedStatement ps = con.prepareStatement(query);
ps.setInt(1, 101);
ResultSet rs = ps.executeQuery();
while(rs.next()){
    System.out.print("\nName: " + rs.getString("name"));
}
```

Inserting Data Example

```
String query = "INSERT INTO students(id, name) VALUES(?,?)";
PreparedStatement ps = con.prepareStatement(query);
ps.setInt(1, 101);
Ps.setString(2, "Shiv");
int result = ps.executeUpdate();
System.out.print("\n"+result + " row(s) inserted.");
```

Updating and deleting data

Update:

```
String query = "UPDATE students SET name=? WHERE id=?";
PreparedStatement ps = con.prepareStatement(query);
ps.setString(1, "John Smith");
ps.setInt(2, 104);
ps.executeUpdate();
```

Delete:

```
String query = "DELETE students WHERE id=?";
PreparedStatement ps = con.prepareStatement(query);
ps.setInt(1, 104);
ps.executeUpdate();
```

Handling Exceptions

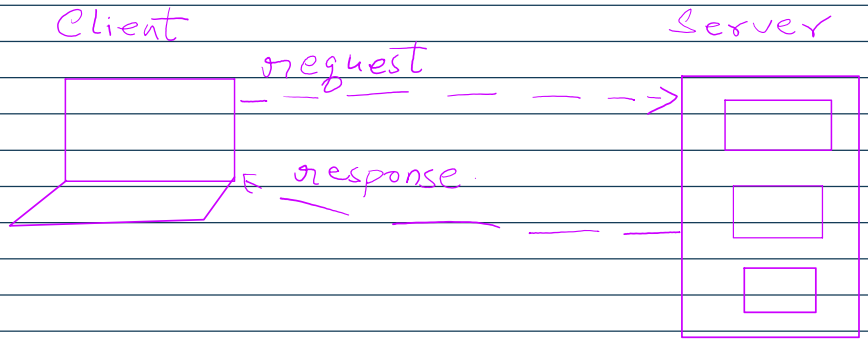
Always use try-catch blocks to handle exceptions gracefully.

```
try{
   //JDBC logic here
}catch(SQLException e){
   System.out.println("Database error: " + e.getMessage());
}
```

Introduction to Socket Programming

16 April 2025 19:52

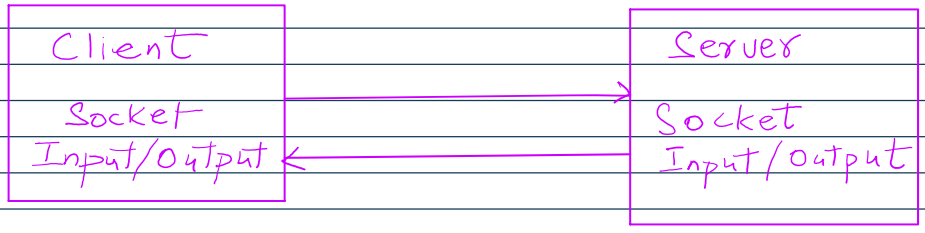
Socket Programming in Java:
Java socket programming is used for communication between the application that is running on different JRE. It can be either connection-oriented or connection-less.



What is socket?
A socket is one endpoint of a 2 way communication between two programs running on the network. The socket is bound to a port number so that TCP layer can layer can identify the application that data. is destined to be sent.

Client Socket: initiates communication.
Server Socket: Weights and listen for incoming requests.

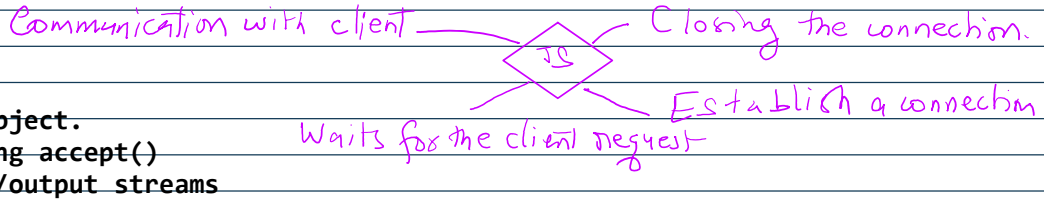
Both client and server use the "java.net" package to establish the connection and communicate.



Methods of socket:

- **public InputStream getInputStream() => returns the InputStream attached with this socket**
- **public OutputStream getOutputStream() => Returns the output stream attached with this socket.**
- **public synchronized void close() => close this socket**
- **public Socket accept() => Returns the socket and establish a connection between client and server.**
- **public synchronized void close() => Closes the server socket.**

Connection Lifecycle:



Server side:

1. **Create a ServerSocket Object.**
2. **Wait for connection using accept()**
3. **Communicate using input/output streams**
4. **Close connection.**

Client side:

1. **Create a socket object specifying server IP address and port number.**

2. Connect to the server.
3. Communicate using input or output streams.
4. Close the connections.

Server code:

```
package SocketProgramming;
import java.io.*;
import java.net.*;
public class Server{
    public static void main(String[] args) {
        try(ServerSocket serverSocket = new ServerSocket(5050)){
            System.out.println("Server is listening on port 5050");
            for(;;){
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected");
                new ClientHandler(clientSocket).start();
            }
        }
        catch(IOException ioex){
            ioex.printStackTrace();
        }
    }
}
class ClientHandler extends Thread{
    private Socket socket;
    public ClientHandler(Socket socket){
        this.socket = socket;
    }
    public void run(){
        try(BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(),true)){
                String text;
                while((text = input.readLine())!= null){
                    System.out.println("Received: " + text);
                    output.println("Server echo: "+ text);
                }
            }
        catch(IOException ioe){
            ioe.printStackTrace();
        }
    }
}
```

Client code:

```
package SocketProgramming;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 5050);
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
```

```
            PrintWriter output = new PrintWriter(socket.getOutputStream(),true);
            BufferedReader serverInput = new BufferedReader(new
    InputStreamReader(socket.getInputStream())) {
                System.out.println("Connected to server. Type messages: ");
                String msg;
                while(!(msg = input.readLine()).equalsIgnoreCase("exit")){
                    output.println(msg);
                    String reply = serverInput.readLine();
                    System.out.println(reply);
                }
            } catch (IOException e) {
                // TODO: handle exception
                e.printStackTrace();
            }
        }
    }
```

```
Server is listening on port 5050
New client connected
Received: Each client gets its own thread.
Received: this is interesting
```

```
Connected to server. Type messages:
Each client gets its own thread.
Server echo: Each client gets its own thread.
this is interesting
Server echo: this is interesting
exit
```

Advanced Feature:

Multithreaded Server

- **Each client gets its own thread, allowing multiple clients to interact simultaneously.**

Serialization

- **Send objects over sockets using ObjectInputStream and ObjectOutputStream.**

Secure Socket layer(SSL)

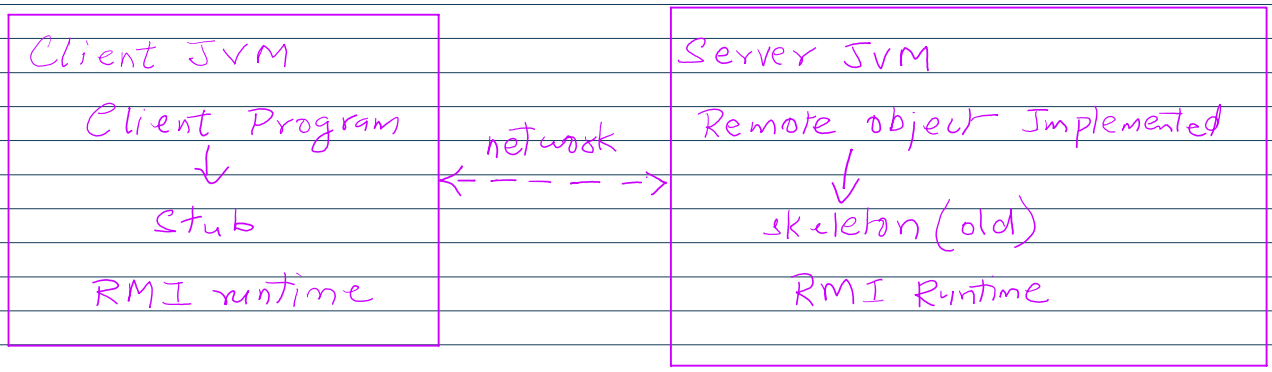
- **Use SSLSocket and SSLServer socket for encrypted communication.**

RMI

16 April 2025 20:52

What is RMI (Remote Method Invocation)?
Java Remote Method Invocation(RMI) is an Java API that performs a mechanism to allow the invocation of methods on remote objects.

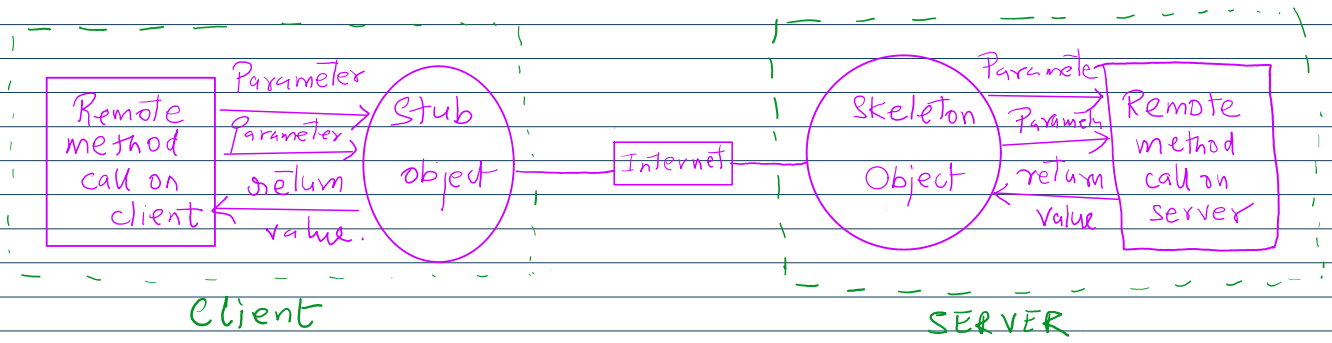
Definition:
Java RMI allows an object running in one Java Virtual Machine (JVM) to invoke methods on an object running in another JVM. This is Java's native way to implement it, distributed object oriented computing. It abstracts complex network communication and serialization, making remote communication as Seamless as local method calls.



Stub: Client-side proxy for the remote object, Handles marshaling (packing data), sending request, and unmarshalling (interpreting responses).
In other words, a stub is a local object that acts as a proxy for a remote object when a client wants to invoke a method on a remote object. It does so through the stub.

Skeleton: Server side component (prior to Java, 1.2.) today. the server side object handles request directly.
In other words, skeleton is used to receive method invocation from clients and forward them to the remote object.

RMI Registry: Acts like a phone book maps. name to remote object.



Stub: Information block → Server
- ID of remote object
- Method name to invoke
- Parameter to pass.

Skeleton.
- calls desired method
- forwards the parameter
- return value to Stub object.

Goals of RMI
(i) Minimize complexity of the Application.

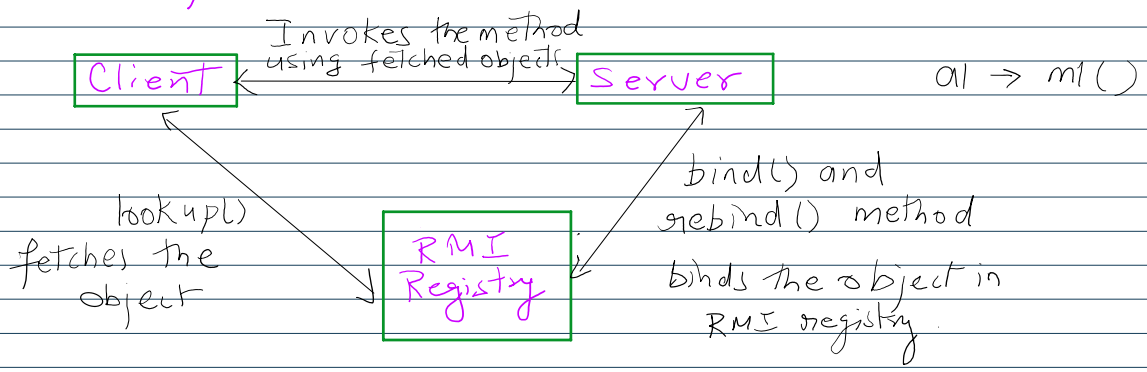
Goals of RMI

(i) Minimize complexity of the Application.

(ii) Distributed Garbage collection.

(iii) Minimizes difference between working with local & remote objects.

RMI Registry : It is a namespace on which all server objects are placed.



How RMI works?

1. **Client makes a method call on a remote interface.**
2. **The stub object:**
   a. **Marshall's method parameters into bytes.**
   b. **Sends data to the server using the network.**
3. **On the server, the RMI runtime:**
   a. **Receives the call and unmarshal the parameters.**
   b. **Invokes the actual method on the server object.**
4. **The method executes on the server and returns a result.**
5. **The stub receives the return value and gives it to the client program.**

```
package RMI;
import java.rmi.Remote;
import java.rmi.RemoteException;
// remote interface
public interface Calculator extends Remote {
    int add(int a, int b) throws RemoteException;
}
```

```
package RMI;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
// implementing Remote Object
public class CalculatorImp extends UnicastRemoteObject implements Calculator{
    protected CalculatorImp() throws RemoteException{
        super();
    }
    public int add(int a, int b){
        return a+b;
    }
}
```

```
package RMI;
import java.rmi.registry.LocateRegistry;
```

```java
import java.rmi.registry.Registry;
public class CalculatorServer {
    public static void main(String[] args) {
        try {
            Calculator skeleton = new CalculatorImp();
            Registry registry = LocateRegistry.createRegistry(1099); //Start RMI registry in code
            registry.rebind("CalcService", skleton);
            System.out.println("Server is running...");
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }
}
```

```java
package RMI;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);
            Calculator stub = (Calculator)registry.lookup("CalcService");
            System.out.println(" 15 + 5 = "+stub.add(15, 5));
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }
}
```

Remote: Marker interface tells. Java that method can be called remotely.
UnicastRemoteObject: Helps export the object to receive incoming calls.
Stub: Auto generated or internal proxy to contact remote object.
Skeleton: Legacy server side proxy now internal.
Registry: Allows client to look up remote objects by name.

Multithreading

16 April 2025 23:13

Process

A process is an independent program in execution. It has its own memory space system resources and at least one thread (main thread).
Example: Running to Java applications means two separate processes.(e.g., game and a calculator)

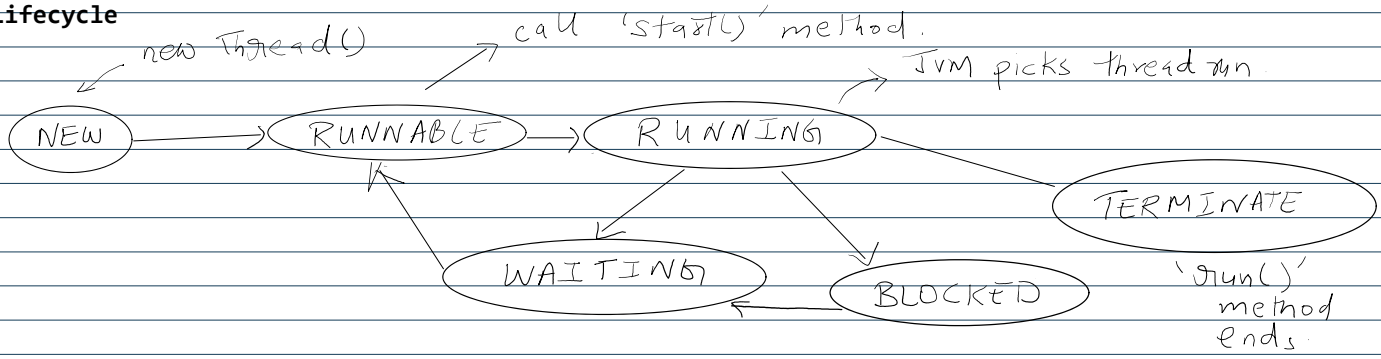
Thread

A thread is the smallest unit of execution within a process. Multiple threads can run concurrently in the same process, sharing memory and resources.
Example: In a web browser (a process) One thread may load the page another, may play a video.

Multiprocessing vs Multithreading

Feature	Muti-processing	Multithreading
Units of execution.	Multiple processes.	Multiple threads.
Memory usage.	More (Each process has separate memory)	Less (threads share memory)
Context switching	Expensive	Faster
Communication	Interprocess communication (IPC) Needed.	Direct sharing of variables.
Stability	More stable (crash isolated)	Less stable(one thread crash may affect others).

Thread Lifecycle



Creating Threads in Java

① By extending Thread class

```
public class MyClass extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}
```

② By implementing Runnable Interface.

```
public class MyRunable implements Runnable. {
    public void run() {
```

public class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable Thread is running...");
    }
}

Usage:

```
Thread t = new Thread(new MyRunnable());
t.start();
```

Common Methods

start() => Starts thread execution.
run() => Code to execute in thread.
sleep(ms) => Pause thread for milliseconds.
join() => Waits for the thread to die.
yield() => Hint to thread scheduler to pause.
isAlive() => Checks if thread is alive.
setPriority() => Sets priority(1 to 10)
getPriority() => Gets priority

Thread Synchronization: When multiple thread access shared data. (Example of bank account.) Bears a risk of data inconsistency. Synchronization ensures only one thread access or critical section at a time.

Synchronized method:

```
class  BankAccount{
   private int balance = 1000;
   public synchronized void withdraw(int amount){
      if(balance >= amount){
         System.out.println(Thread.currentThread().getName() + " is withdrawing " + amount);
         Balance -= amount;
      }
      else{
         System.out.println("insufficient balance for " + Thread.currentThread().getName());
      }
   }
}
```

Synchronized blocks

```
public void withdrawl(int amount){
    synchronized(this){
      // critical section
    }
}
```

Inter thread communication.

1. **wait() => releases locks**
2. **notify() => wakes up a single waiting thread**
3. **notifyAll() => wakes up all waiting thread**

Example:

```
    synchronized(obj){
        while(!condition){
           Obj.wait();
```

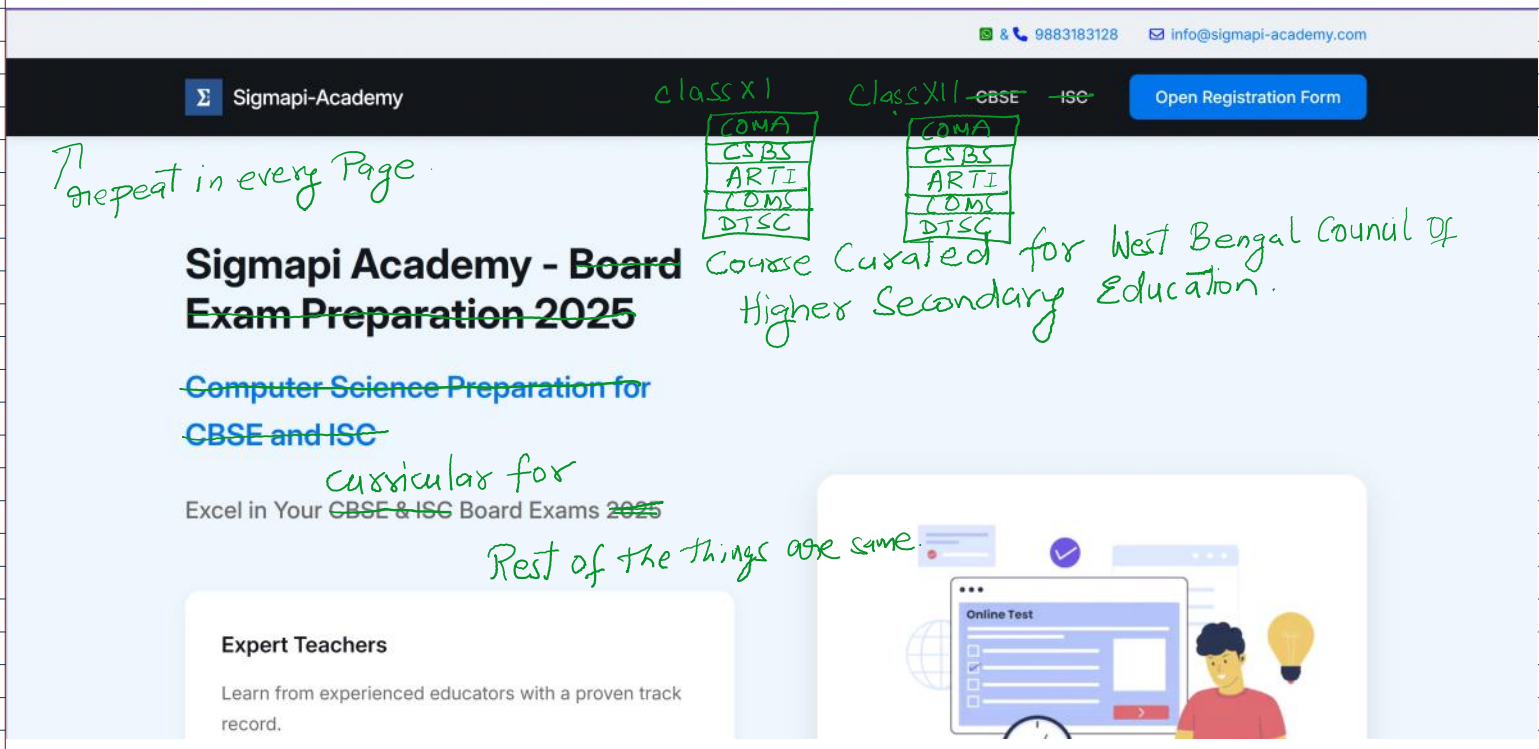
```
                    //do work
                    Obj.notify();
                }
            }
```

Problems:

1. Deadlock
2. Race conditions
3. Starvation: Thread never gets cpu time.

Website discussion

26 May 2025 19:06



class XI: COMA, CSBS, ARTI, COMS, DTSC

Class XII: COMA, CSBS, ARTI, COMS, DTSC

Course Curated for West Bengal Council of Higher Secondary Education.

1. DTSC → XI
2. DTSC → XII

Semester – 1

Computer Fundamentals

image

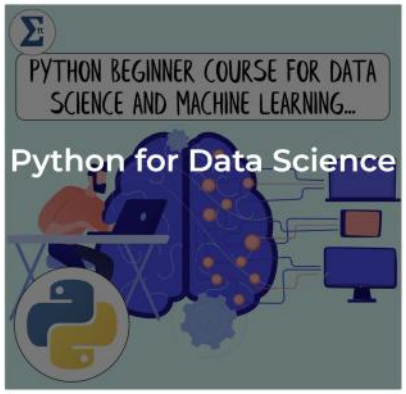
Introduction to Python Programming.

image

History of AI and Introduction to Linear Algebra.

image

Know More → button → a page will open and describe the syllabus in detail.



img

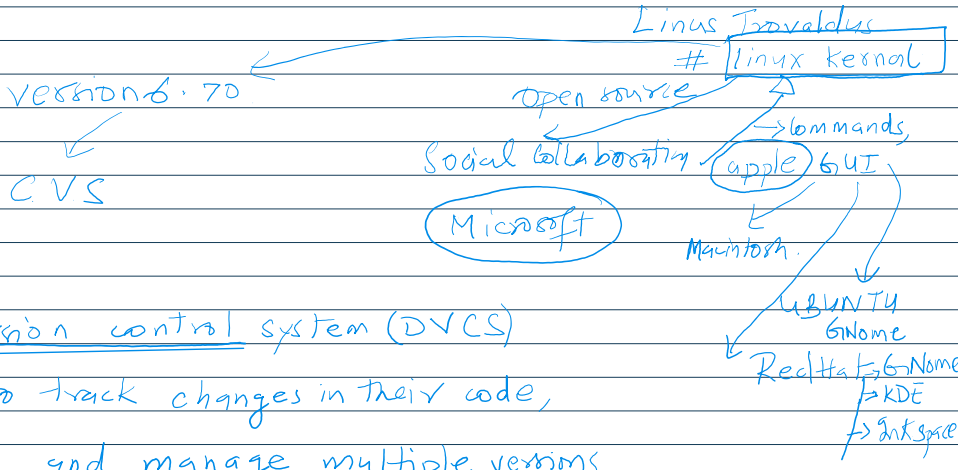
AI & Robotics
ICSE & ISC
Class 9-12

img

AI CBSE
class 9-12

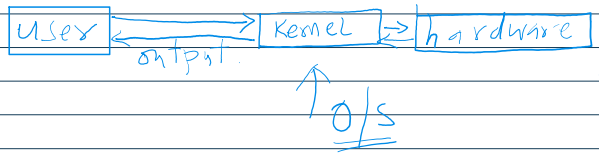
Git Concept

16 June 2025 19:13

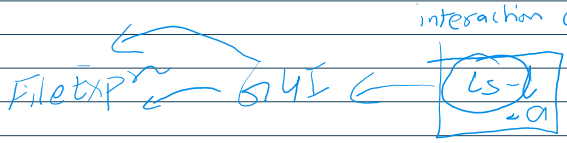


What is Git?

Git is a distributed version control system (DVCS) that allows developers to track changes in their code, collaborate with others, and manage multiple versions of a project efficiently.



Kernel is program where multiple command are stored, to control operating system internal activities and as well user interaction commands



Version control :- Git helps mange different versions of code, so you can :

Go back to previous version.

View the history of changes.

See who changed what and when.

Snapshots, not difference : Git stores data as snapshots of entire project rather than line-by-line changes.

Distributed system :- Every developer has a full copy of the project with complete history — not just a central copy.

Important Git Terminologies

Term	Description

Repository(repo)	A folder. Containing all project files and their history tracked by git.
Commit	A snapshot of changes with a unique ID.
Branch	A parallel version of the repo to work on features without affecting the main code.
Merge	Combines changes from one branch into another.
Clone	Makes a local copy of a remote repository.
Pull	Fetch and integrate changes from the remote repository.
Push	Upload your local changes to the remote repository.

Basic Git Commands