

Java

Collections Framework



COLLECTIONS

-> Primitive Data Types are used to store single value into a variable

```
int x = 10;
```

```
int y = 20;
```

```
int z = 30
```

-> If we want to store 1000 values, then we need to declare 1000 variables in program which is not at all recommended.

-> To store multiple values into single variable then we can use Arrays concept.

-> Array is an collection of fixed number of homogeneous data elements

OR

-> An array represents a group of elements of same data type.

-> The main advantage of array is we can represent huge number of elements by using single variable. So, the readability of the code is improved.



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

```
arr[0] = 100;
```

```
arr[1] = 200;
```

```
..
```

```
arr[4] = 500;
```

Limitations of array:

-> Arrays are fixed in size that is once we create an array there is no chance of increasing or decreasing the size of array based on our requirement. Hence to use array concept we must know the size in advance which may not possible every time.

-> Arrays can hold only homogeneous data elements.

Example:

```
Car c = new Car [100];
```

```
c[0] = new Car(); // Valid
```

```
c[1] = new Bus(); //Invalid(Compile time Error)
```

-> We can resolve this problem by using Object type Array (Object [])

Example:

```
Object[] o=new Object[100];  
o[0]=new Car();  
o[1]=new Bus();
```

-> Arrays concept is not implemented based on some data structure hence we cannot expect ready-made method support.

-> For every requirement we have to write the code explicitly like insert, retrieve, update, sort etc..

To overcome the above limitations, we should go for collection concept.

-> Collections are growable in nature that is based on our requirement we can increase or decrease the size hence memory point of view collections concept is recommended to use.

-> Collections 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. As a programmer we can use these methods directly without writing the functionality on our own.

#	Array	Collection
1	Arrays are fixed in size	Collections are growable in nature
2	With respect to memory, Arrays are not recommended to use	With respect to memory, Collections are recommended to use
3	With respect to performance Arrays are recommended to use	With respect to performance Collections are not recommended to use
4	Arrays can hold only homogeneous datatype elements	Collections can hold homogeneous & heterogeneous datatype elements
5	No underlying data structure. Hence no readymade method support	Based on standard data structure. Hence it has readymade method support
6	It can hold both Primitives and Objects	It can hold only Objects

Collection: If we want to represent a group of objects as single entity then we should go for Collections.

Collection framework: It defines several classes and interfaces to represent a group of Objects as single entity.

Q) What is the difference between Collection and Collections?

Ans) Collection is an interface which can be used to represent a group of Objects as a single entity whereas Collections is an utility class present in java.util package to define several utility methods for Collection Objects.

- Collection is an interface
- Collections is a Class

-> All the collection classes are available in "java.util" (utility) package.

-> All the collection interfaces and collection class and together as collection frame work.

-> All the collection classes are classified into three categories

- 1) List
- 2) Set
- 3) Queue
- 4) Map

1. List:

- This category is used to store group of individual elements where the elements can be duplicated.

- List is an Interface whose object can not be created directly.

- To work with this category we have to use following implementations class of list interface

Ex: ArrayList, Linked list, Vector, Stack

2.Set:

- This category is used to store a group of individual elements. But they elements can't be duplicated.

- Set is an interface whose object cannot be created directly.

- To work with this category, we have to use following implementations class of Set interface

Ex: HashSet, LinkedHashSet and TreeSet

3. Queue

-> This category is used to hold the elements about to be processed in FIFO(First In First Out) order.

-> It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle

Ex: PriorityQueue, BlockingQueue

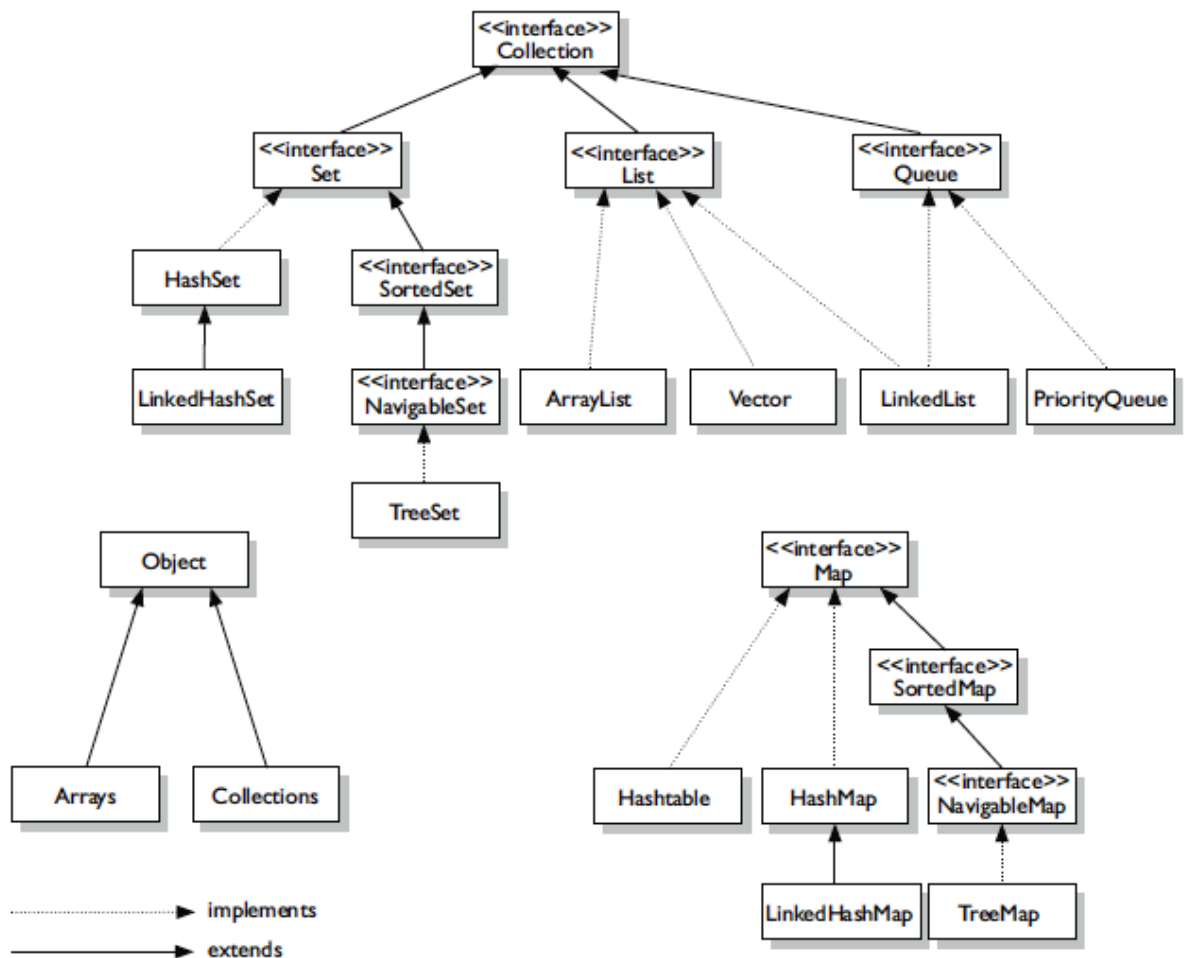
4. Map:

- This category is used to store the element in the form key value pairs where the keys can't be duplicated, values can be duplicated.

- Map is an interface whose object cannot be created directly.
- To work with this category, we have to use following implementation classes of Map interface

Ex: HashMap, LinkedHashMap, TreeMap, Hashtable

Collections Hierarchy



List interface

- > It is the child interface of Collection
- > If we want to represent a group of individual objects where duplicates are allowed and insertion order is preserved. Then we should go for List.
- > We can differentiate duplicate Objects and we can maintain insertion order by means of index hence "index plays important role in List"

List interface defines the following specific methods.

1)boolean add(int index,Object o);

- 2)boolean addAll(int index,Collection c);
- 3)Object get(int index);
- 4)Object remove(int index);
- 5)Object Set(int index,Object o);//to replace
- 6)int indexOf(Object o); //Returns index of first occurrence of o
- 7)int lastIndexOf(Object o);
- 8)ListIterator listIterator();

ArrayList:

- > ArrayList is an implementation class of Collection interface
- >The underlying data structure is resizable (Internally it will use Array to store data)
- > Duplicate Objects are allowed
- > Insertion order is preserved
- > Heterogeneous Objects are allowed
- > Null insertion is possible

ArrayList Constructors:**1) ArrayList al=new ArrayList();**

- > It creates an empty ArrayList Object with default initial capacity 10
- > If ArrayList reaches its maximum capacity then a new ArrayList Object will be created with

$$\text{NewCapacity} = (\text{Current Capacity} * 3/2) + 1$$

2) ArrayList al=new ArrayList(int initialCapacity);

- > Creates an empty ArrayList Object with the specified initial capacity.

3) ArrayList al=new ArrayList(Collection c);

- > Creates an equivalent ArrayList Object for the given Collection that is this constructor meant for inter conversation between Collection Objects.

-> ArrayList and Vector classes implements RandomAccess interface so that any random element we can access with the same speed.

-> RandomAccess interface present in util package and doesn't contain any methods. It is a marker interface.

```
ArrayListDemo.java
1 package in.ashokit;
2
3 import java.util.ArrayList;
4
5 public class ArrayListDemo {
6
7     public static void main(String[] args) {
8
9         ArrayList al = new ArrayList();
10        al.add("ashokit");
11        al.add(101);
12        al.add(202.05);
13        System.out.println(al); // [ashokit, 101, 202.05]
14        al.remove(1);
15        System.out.println(al); // [ashokit, 202.05]
16        al.add("hyd");
17        al.add("java");
18        System.out.println(al); // [ashokit, 202.05, hyd, java]
19    }
20 }
21
```

LinkedList

- > LinkedList is one of the implementation classes of Collection interface
- > The underlying data structure is double LinkedList
- > If our frequent operation is insertion or deletion in the middle then LinkedList is the best choice
- > If our frequent operation is retrieval then LinkedList is not best option
- > Duplicate Objects are allowed
- > Insertion order is preserved
- > Heterogeneous Objects are allowed
- > NULL insertion is possible
- > Implements Serializable and Cloneable interfaces but not RandomAccess

Note: Usually we can use linked list to implement Stacks and Queues to provide support for this requirement LinkedList class defines the following 6 specific methods.

- 1) void addFirst(Object o);
- 2) void addLast(Object o);
- 3) Object getFirst();
- 4) Object getLast();

5) Object removeFirst();

6) Object removeLast();

LinkedList Constructors:

1) LinkedList l=new LinkedList();

It creates an empty LinkedList Object.

2) LinkedList l=new LinkedList(Collection c);

To create an equivalent LinkedList Object for the given Collection.

```
LinkedListDemo.java
1 package in.ashokit;
2
3 import java.util.LinkedList;
4
5 public class LinkedListDemo {
6     public static void main(String[] args) {
7         LinkedList ll = new LinkedList();
8         ll.add("ashokit");
9         ll.add(40);
10        ll.add(null);
11        ll.add("ashokit");
12        System.out.println(ll); // [ashokit,40,null,ashokit]
13        ll.add(0, "java");
14        System.out.println(ll); // [java,ashokit,40,null,ashokit]
15        ll.set(0, "oracle");
16        System.out.println(ll); // [oracle,ashokit,40,null,ashokit]
17        ll.removeLast();
18        System.out.println(ll); // [oracle,java,40,null]
19        ll.addFirst("ashok");
20        System.out.println(ll); // [ashok,oracle,java,40,null]
21    }
22 }
23
```

IQ: what is the diff b/w ArrayList and LinkedList

-> ArrayList is slower in insertion and deletion of elements because it internally requires shifting operations, But faster in accessing the elements because ArrayList use index position for every element.

-> LinkedList is faster in insertion and deletion of elements because it just require modifying the links of nodes instead of shifting operations, But slower in accessing the elements because LinkedList does not use any index position.

Vector

-> Vector is the implementation class of List interface which is also used to store group of individual objects where duplicate values are allowed

-> Vector is exactly similar to ArrayList but ArrayList is not a synchronized class where Vector is a synchronized class

-> Vector is also called legacy class because it is available from java 1.0 version.

Vector Class Constructors

- 1) `Vector<E> v = new Vector<E>();`
- 2) `Vector<E> v = new Vector<E>(int capacity);`
- 3) `Vector<E> v = new Vector<E>(Collection obj);`

```
VectorDemo.java ✕
1 package in.ashokit;
2
3 import java.util.Vector;
4
5 public class VectorDemo {
6
7     public static void main(String[] args) {
8         Vector<String> v = new Vector<String>();
9         v.add("sachin");
10        v.add(new String("sehwagh"));
11        v.add("kohli");
12        v.add("dhoni");
13        v.add("suresh");
14        System.out.println(v);
15        System.out.println(v.size());
16    }
17 }
18
```

Stack

- > Stack is a child class of Vector and implements List interface
- > Stack stores a group of objects by using a mechanism called LIFO
- > LIFO stands for Last in first out, it means last inserted element deleted first.

Stack Class Constructor:

```
Stack<E> s = new Stack<E>( );
```

Methods:

- > We can use all collection Methods
 - > We can also use legacy methods of Vector class like addElement(), removeElement(), setElementAt(),.....
 - > But if we want to follow the LIFO mechanism, we should use Stack methods like follows
1. E push(E obj) : this method will add new element into the Stack
 2. E pop() : this method deletes the top element available on Stack
 3. E peek() : this method just returns the top element available on Stack

```
StackDemo.java
1 package in.ashokit;
2
3 import java.util.Stack;
4
5 public class StackDemo {
6
7     public static void main(String[] args) {
8         Stack<Double> s = new Stack<Double>();
9         s.push(10.2);
10        s.push(50.2);
11        s.push(30.2);
12        s.push(40.2);
13        s.push(70.2);
14        System.out.println(s); // [10.2, 50.2, 30.2, 40.2, 70.2]
15        System.out.println(s.pop()); // 70.2
16        System.out.println(s); // [10.2, 50.2, 30.2, 40.2]
17        System.out.println(s.peek()); // 40.2
18        System.out.println(s); // [10.2, 50.2, 30.2, 40.2]
19    }
20 }
21
```

Cursors of Collection Framework

-> cursors are mainly used to access the elements of any collection

-> we have following 3 types of cursors in Collection Framework

- 1.Iterator
- 2.ListIterator
- 3.Enumeration

Iterator

-> this cursor is used to access the elements in forward direction only

-> this cursor can be applied Any Collection (List, Set)

-> while accessing the methods we can also delete the elements

-> Iterator is interface and we cannot create an object directly.

-> if we want to create an object for Iterator, we have to use iterator () method

Creation of Iterator:

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

here iterator() method internally creates and returns an object of a class which implements Iterator interface.

Methods

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

```
ArrayListDemo.java
1 package in.ashokit;
2
3 import java.util.ArrayList;
4 import java.util.Iterator;
5
6 public class ArrayListDemo {
7
8     public static void main(String[] args) {
9
10         ArrayList<String> al = new ArrayList<>();
11         al.add("ashok");
12         al.add("it");
13         al.add("java");
14         al.add("training");
15
16         Iterator<String> iterator = al.iterator();
17
18         while (iterator.hasNext()) {
19             String next = iterator.next();
20             System.out.println(next);
21         }
22     }
23 }
```

**ASHOK IT***Learn Here.. Lead Anywhere..!!*

2. ListIterator

-> This cursor is used to access the elements of Collection in both forward and backward directions

-> This cursor can be applied only for List category Collections

-> While accessing the methods we can also add, set, delete elements

-> ListIterator is interface and we can not create object directly.

-> If we want to create an object for ListIterator we have to use listIterator() method

creation of ListIterator:

ListIterator<E> it = l.listIterator();

here listIterator() method internally creates and returns an object of a class which implements ListIterator interface.

Methods

1. boolean hasNext();
2. Object next();
3. boolean hasPrevious();

Java

4. Object previous();
5. int nextIndex();
6. int previousIndex();
7. void remove();
8. void set(Object obj);
9. void add(Object obj);

```
ArrayListDemo.java
1 package in.ashokit;
2
3 import java.util.ArrayList;
4 import java.util.ListIterator;
5
6 public class ArrayListDemo {
7
8     public static void main(String[] args) {
9
10         ArrayList<String> al = new ArrayList<>();
11         al.add("ashok");
12         al.add("it");
13         al.add("java");
14         al.add("training");
15
16         ListIterator<String> listIterator = al.listIterator();
17
18         while (listIterator.hasNext()) {
19             String next = listIterator.next();
20             System.out.println(next);
21         }
22     }
23 }
```

3. Enumeration

- > this cursor is used to access the elements of Collection only in forward direction
- > this is legacy cursor can be applied only for legacy classes like Vector, Stack, Hashtable.
- > Enumeration is also an interface and we can not create object directly.
- > If we want to create an object for Enumeration we have to use a legacy method called elements() method

Creation of Enumeration:

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

here elements() method internally creates and returns an object of a class which implements Enumeration interface.

Methods

1. boolean hasMoreElements()
2. Object nextElement();

```

VectorDemo.java
1 package in.ashokit;
2
3 import java.util.Enumeration;
4 import java.util.Vector;
5
6 public class VectorDemo {
7
8     public static void main(String[] args) {
9         Vector<Integer> v = new Vector<Integer>();
10        v.add(10);
11        v.add(25);
12        v.add(50);
13        v.add(20);
14        v.add(25);
15        v.add(23);
16        v.add(25);
17        System.out.println(v);
18
19        Enumeration e = v.elements();
20        while (e.hasMoreElements()) {
21            System.out.print(e.nextElement() + " ");
22        }
23    }
24 }
25

```

	Iterator	ListIterator	Enumeration
Applicable to	List, Set	List implemented classes	legacy classes
Accessing Direction	only forward	both forward & backward	only forward
Operations	access, remove	access, remove, add, set	only accessing
method to create	iterator()	listIterator()	elements()
legacy	✗	✗	✓
No. of Methods	3	9	2

Set category**HashSet**

-> HashSet is the implementation class of Set interface which is also used to store group of individual objects but duplicate values are not allowed

-> HashSet internally follows hashtable structure where all the elements are stored using hashing technique which will improve the performance by reducing the waiting time.

-> HashSet is not a synchronized class

-> HashSet supports only one null value.

-> HashSet is called unordered Collection because it is not guarantee for insertion order of elements.

creation of HashSet:

```
HashSet<E> hs = new HashSet<E>();
```

```
HashSet<E> hs = new HashSet<E>(int capacity);
```

```
HashSet<E> hs = new HashSet<E>(int capacity, float loadfactor);
```

```
HashSet<E> hs = new HashSet<E>(Collection obj);
```

Methods

1. boolean add(E obj)
2. boolean remove(E obj)
3. int size()
4. void clear()
5. boolean contains(E obj)
6. boolean isEmpty()

```
HashSetDemo.java
1 package in.ashokit;
2
3 import java.util.HashSet;
4 import java.util.Iterator;
5
6 public class HashSetDemo {
7
8     public static void main(String ar[]) {
9         HashSet<Integer> hs = new HashSet<Integer>();
10        hs.add(27);
11        hs.add(12);
12        hs.add(null);
13        hs.add(9);
14        hs.add(21);
15        System.out.println(hs.size());
16        for (Integer e : hs) {
17            System.out.println(e);
18        }
19        Iterator it = hs.iterator();
20        while (it.hasNext()) {
21            System.out.println(it.next());
22        }
23    }
24 }
```

LinkedHashSet

-> LinkedHashSet is the implementation class of Set interface which is also used to store group of individual objects but duplicate values are not allowed

-> LinkedHashSet internally follows hashtable + doubly linked list structures

-> LinkedHashSet is not a synchronized class

-> LinkedHashSet supports only one null value.

-> LinkedHashSet is called as ordered Collection because it is guarantee for insertion order of elements.

Creation of LinkedHashSet:

```
LinkedHashSet<E> hs = new LinkedHashSet<E>();
```

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(int capacity);
```

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(int capacity,float loadfactor);
```

```
LinkedHashSet<E> hs = new LinkedHashSet<E>(Collection obj);
```



```
1 package in.ashokit;
2
3 import java.util.Iterator;
4 import java.util.LinkedHashSet;
5
6 public class LinkedHashSetDemo {
7
8     public static void main(String ar[]) {
9         LinkedHashSet<Integer> hs = new LinkedHashSet<Integer>();
10        hs.add(27);
11        hs.add(12);
12        hs.add(null);
13        hs.add(9);
14        hs.add(21);
15        System.out.println(hs.size());
16        for (Integer e : hs) {
17            System.out.println(e);
18        }
19        Iterator it = hs.iterator();
20        while (it.hasNext()) {
21            System.out.println(it.next());
22        }
23    }
24 }
25
```

SortedSet:

- 1) It is child interface of Set
- 2) If we want to represent a group of "unique Objects" according to some sorting order then we should go for SortedSet interface.
- 3) That sorting order can be either default natural sorting order OR customized sorting order.

SortedSet interface defines the following 6 specific methods

- 1) Object first();
- 2) Object last();
- 3) SortedSet headSet(Object o);

It returns the elements whose elements are < o

- 4) SortedSet tailSet(Object o);

It returns the elements whose elements are >= o

- 5) SortedSet subSet(Object o1, Object o2);

It returns the elements whose elements are >=o1 and < o2

- 6) Comparator comparator();

Returns the comparator Object that describes underlying sorting technique. If we follow default natural sorting order then this method returns null.

```
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetDemo {

    public static void main(String[] args) {

        SortedSet ss=new TreeSet();

        for(int i=10;i<=20;i++)

            ss.add(i);

        System.out.println(ss.first());//10

        System.out.println(ss.last());//20

        System.out.println(ss.headSet(16));//[10, 11, 12, 13, 14, 15]

        System.out.println(ss.tailSet(18));//[18, 19, 20]

        System.out.println(ss.subSet(12,17));//[12, 13, 14, 15, 16]

        System.out.println(ss.comparator());//null

    }
}
```

TreeSet

- > TreeSet is the implementation class of Set interface which is also used to store group of individual objects but duplicate values are not allowed
- > TreeSet internally follows tree structure
- > TreeSet is not a synchronized class
- > TreeSet is called as unordered Collection because it is not guarantee for insertion order of elements but all elements are stored in sorted order(by default ascending order)
- > TreeSet supports only one null value if TreeSet is Empty otherwise TreeSet does not support null values because it internally perform comparison operations but we never compare a null value. with any object and it will throw a RuntimeException saying NullPointerException
- > TreeSet does not allow to store different types of objects because it internally perform comparison operations but we never compare a 2 different types of it will throw a RuntimeException saying ClassCastException

Creation of TreeSet

```
TreeSet<E> ts = new TreeSet<E>();
```

```
TreeSet<E> ts = new TreeSet<E>(SortedSet);
```

```
TreeSet<E> ts = new TreeSet<E>(Comparator);
```

```
TreeSet<E> ts = new TreeSet<E>(Collection obj)
```

```
TreeSetDemo.java
1 package in.ashokit;
2
3 import java.util.TreeSet;
4
5 public class TreeSetDemo {
6
7     public static void main(String[] args) {
8         TreeSet<String> ts = new TreeSet<>();
9
10        ts.add("java");
11        ts.add("programming");
12        ts.add("language");
13
14        // ts.add(10); // invalid
15
16        System.out.println(ts);
17    }
18 }
19
```

NULL acceptance:

For the empty TreeSet as the first element "null" insertion is possible but after inserting that null if we try to insert any other value then we will get NullPointerException.

For the non-empty TreeSet if we try to insert null then we will get NullPointerException.

```
TreeSetDemo.java
1 package in.ashokit;
2
3 import java.util.TreeSet;
4
5 public class TreeSetDemo {
6
7     public static void main(String[] args) {
8         TreeSet ts = new TreeSet();
9         ts.add(new StringBuffer("Java")); // ClassCastException
10        ts.add(new StringBuffer("DevOps"));
11        ts.add(new StringBuffer("MySQL"));
12        ts.add(new StringBuffer("Python"));
13        System.out.println(ts);
14    }
15 }
16
```

String class and all wrapper classes implements Comparable interface but StringBuffer class does not implement Comparable interface hence in the above program we will get ClassCastException.

Comparable interface:

Comparable interface present in java.lang package and contains only one method compareTo() method.

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

Example: obj1.compareTo(obj2);

It returns -ve if obj1 comes before obj2

It returns +ve if obj2 comes before obj1

It returns 0 if obj1 and obj2 are equal

```
CompareTo.java ✕
1 package in.ashokit;
2
3 public class CompareTo {
4     public static void main(String[] args) {
5         System.out.println("A".compareTo("Z")); // -25
6         System.out.println("z".compareTo("a")); // 25
7         System.out.println("A".compareTo("A")); // 0
8     }
9 }
```

If we depend on default natural sorting order then internally JVM will use compareTo() method to arrange Objects in sorting order.

```
CompareTo.java
1 package in.ashokit;
2
3 import java.util.TreeSet;
4
5 public class CompareTo {
6     public static void main(String[] args) {
7         TreeSet<Integer> ts = new TreeSet<>();
8         ts.add(21);
9         ts.add(12);
10        ts.add(1);
11        ts.add(0);
12        ts.add(15);
13        ts.add(17);
14        System.out.println(ts); // [0, 1, 12, 15, 17, 21]
15    }
16 }
```

We can define our own customized sorting by Comparator Object.

Comparable meant for default natural sorting order.

Comparator meant for customized sorting order.

Comparator interface:

Comparator interface present in java.util package. It defines the following two methods.

1) public int compare(Object o1, Object o2);

It returns -ve if o1 comes before o2

It returns +ve if o1 comes after o2

It returns 0 if o1 and o2 are equal

2) public boolean equals(Object o);

Whenever we are implementing comparator interface we have to provide implementation only for compare() method.

Implementing equals() method is optional because it is already available from Object class through inheritance.

Requirement:- Write a program to insert integer Objects into the TreeSet where the sorting order is descending order.

Java

Example:

```
import java.util.Comparator;

import java.util.TreeSet;

public class TreeSetComparator {

    public static void main(String[] args) {

        TreeSet ts=new TreeSet(new MyComparator());//---->1

        ts.add(10);

        ts.add(0);

        ts.add(15);

        ts.add(5);

        ts.add(20);

        System.out.println(ts);//[20, 15, 10, 5, 0]

    }

}

class MyComparator implements Comparator {

    public int compare(Object o1,Object o2) {

        Integer i1=(Integer)o1;

        Integer i2=(Integer)o2;

        if(i1<i2)

            return 1;

        else if(i1>i2)

            return -1;

        else

            return 0;

    }

}
```

Comparable vs Comparator

For predefined Comparable classes default natural sorting order is already available, if we are not satisfied with default natural sorting order then we can define our own customized sorting order by Comparator.

For predefined non Comparable classes (Like StringBuffer) default natural sorting order is not available, we have to define our own sorting order by using Comparator Object.

For our own classes (Like Customer, Student and Employee) we can define default natural sorting order by using Comparable interface. The person who is using our class, if he is not satisfied with default natural sorting order then he can define his own sorting order by using Comparator Object.

Example:

```
import java.util.Comparator;
```

```
import java.util.TreeSet;
```

```
class Employee implements Comparable {
```

```
String name;
```

```
int eid;
```

```
Employee(String name,int eid) {
```

```
    this.name=name;
```

```
    this.eid=eid;
```

```
}
```

```
public String toString(){
```

```
    return name+"-"+eid;
```

```
}
```

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

```
    int eid1=this.eid;
```

```
    int eid2=((Employee)o).eid;
```

```
    if(eid1 < eid2)
```

```
        return -1;
```

```
    else if(eid1>eid2)
```

```
Java
    return 1;

else
    return 0;

}

}

public class Comp {

    public static void main(String[] args) {

        Employee e1=new Employee("Raju",100);
        Employee e2=new Employee("Rani",101);
        Employee e3=new Employee("John",102);
        Employee e4=new Employee("Smith",103);
        Employee e5=new Employee("Ashok",104);

        TreeSet t=new TreeSet();

        t.add(e1);
        t.add(e2);
        t.add(e3);
        t.add(e4);
        t.add(e5);

        System.out.println(t);

        TreeSet t2=new TreeSet(new MyComparator());

        t2.add(e1);
        t2.add(e2);
        t2.add(e3);
        t2.add(e4);
        t2.add(e5);

        System.out.println(t2);

    }

}
```



```
class MyComparator implements Comparator {  
    public int compare(Object o1, Object o2) {  
        Employee e1 = (Employee) o1;  
        Employee e2 = (Employee) o2;  
        String s1 = e1.name;  
        String s2 = e2.name;  
        return s1.compareTo(s2);  
    }  
}
```

Comparable in Java	Comparator in Java
Comparable interface is used to sort the objects with natural ordering.	Comparator in Java is used to sort attributes of different objects.
Comparable interface compares "this" reference with the object specified.	Comparator in Java compares two different class objects provided.
Comparable is present in java.lang package.	A Comparator is present in the java.util package.
Comparable affects the original class, i.e., the actual class is modified.	Comparator doesn't affect the original class
Comparable provides compareTo() method to sort elements.	Comparator provides compare() method, equals() method to sort elements.

Map category

Map interface is not child interface of Collection and hence we cannot apply Collection interface methods. Map interface defines the following specific methods.

- 1) Object put(Object key, Object value);
- 2) void putAll(Map m);
- 3) Object get(Object key);
- 4) Object remove(Object key);
- 5) boolean containsKey(Object key);

Java

- 6) boolean containsValue(Object value);
- 7) boolean isEmpty();
- 8) int size();
- 9) void clear();
- 10) Set keySet(); //We will get the set of keys
- 11) Collection values(); //We will get the set of values
- 12) Set entrySet(); //We will get the set of entryset

HashMap

-> HashMap is the implementation class of Map interface which is used to store group of objects in the form of Key-Value pairs where but Keys cannot be duplicated but values can be duplicated

-> HashMap internally follows hashtable data structure

-> HashMap is not a synchronized class

-> HashMap supports only one null value for Key Objects but we can store multiple null values for Value Object

-> HashMap is called unordered Map because it is not guarantee for insertion order of elements.

creation of HashMap:

```
HashMap<K,V> hm = new HashMap<K,V>();
```

```
HashMap<K,V> hm = new HashMap<K,V>(int capacity);
```

```
HashMap<K,V> hm = new HashMap<K,V>(int capacity, float loadfactor);
```

```
HashMap<K,V> hm = new HashMap<K,V>(Map obj);
```

```
MapDemo.java
4
5 public class MapDemo {
6
7     public static void main(String[] args) {
8
9         HashMap<Integer, String> hm = new HashMap<Integer, String>();
10
11         hm.put(11, "sachin");
12         hm.put(25, "kohli");
13         hm.put(12, "yuvaraj");
14
15         System.out.println(hm);
16         System.out.println(hm.size());
17
18         Set<Integer> ks = hm.keySet();
19         System.out.println(ks);
20
21         Collection<String> cv = hm.values();
22         System.out.println(cv);
23
24         Set<?> entry = hm.entrySet();
25         System.out.println(entry);
26
27         System.out.println(hm.containsKey(12));
28
29         System.out.println(hm);
30     }
31 }
32
```

Learn Here.. Lead Anywhere..!!

LinkedHashMap

- > LinkedHashMap is the implementation class of Map interface which is also used to store group of 3 objects in the form of Key-Value pairs where Keys can't be duplicated but values can be duplicated
- > LinkedHashMap internally follows Hashtable + doubly linked list structures
- > LinkedHashMap is not a synchronized class
- > LinkedHashMap supports only one null value for Key Objects but we can store multiple null values for Value Object
- > LinkedHashMap is called as ordered Map because it is guarantee for insertion order of elements.

SortedMap:

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

-> SortedMap interface defines the following 6 specific methods.

- 1) Object firstKey();
- 2) Object lastKey();
- 3) SortedMap headMap(Object key);
- 4) SortedMap tailMap(Object key);
- 5) SortedMap subMap(Object key1, Object key2);
- 6) Comparator comparator();

TreeMap

-> TreeMap is the implementation class of Map interface which is also used to store group of objects in the form of Key-Value pairs where Keys can't be duplicated but values can be duplicated.

-> TreeMap internally follows tree structure

-> TreeMap is not a synchronized class

-> TreeMap is called as unordered Map because it is not guarantee for insertion order of elements, but. all elements are

Learn Here.. Lead Anywhere..!!

Hashtable

-> Hashtable is the implementation class of Map interface which is also used to store group of objects in the form of Key-Value pairs where Keys can't be duplicated but values can be duplicated

-> Hashtable is exactly similar to HashMap but Hashtable is a synchronized class where HashMap is not a synchronized class

-> Hashtable does not support null values for both Keys and Values

Constructors:

- 1) Hashtable ht=new Hashtable();

It creates an empty Hashtable Object with default initial capacity 11 and default fill ratio 0.75

- 2) Hashtable ht=new Hashtable(int initialCapacity);
- 3) Hashtable ht=new Hashtable(int initialCapacity, float fillRatio);
- 4) Hashtable ht=new Hashtable(Map m);

```
HashTable.java
1 package in.ashokit;
2
3 import java.util.Enumeration;
4 import java.util.Hashtable;
5
6 public class Hashtable {
7
8     public static void main(String[] args) {
9         Hashtable<String, Integer> ht = new Hashtable<String, Integer>();
10        ht.put("sachin", 200);
11        ht.put("rohit", 264);
12        ht.put("sehwagh", 219);
13        ht.put("ganguly", 183);
14        ht.put("dhoni", 183);
15        System.out.println(ht);
16        Enumeration e = ht.keys();
17        while (e.hasMoreElements()) {
18            System.out.println(e.nextElement());
19        }
20    }
21 }
22 }
23 }
```

IdentityHashMap

It is exactly same as HashMap except the following differences.

- 1) In the case of HashMap JVM will always use equals() method to identify duplicate keys.
- 2) But in the case of IdentityHashMap JVM will use == (double equal to operator) to identify duplicate keys.

Example:

```
import java.util.HashMap;

public class HashMapExample {

    public static void main(String[] args) {

        HashMap hm=new HashMap();

        Integer i1=new Integer(10);

        Integer i2=new Integer(10);

        hm.put(i1,"John");

        hm.put(i2,"Smith");

        System.out.println(hm);//{10=Smith}

    }

}
```

In the above program i1 and i2 are duplicate keys because i1.equals(i2) returns true.

In the above program if we replace HashMap with IdentityHashMap then i1 and i2 are not duplicate keys because i1==i2 returns false. In this case the output is as below.

Here 10==i1 is false

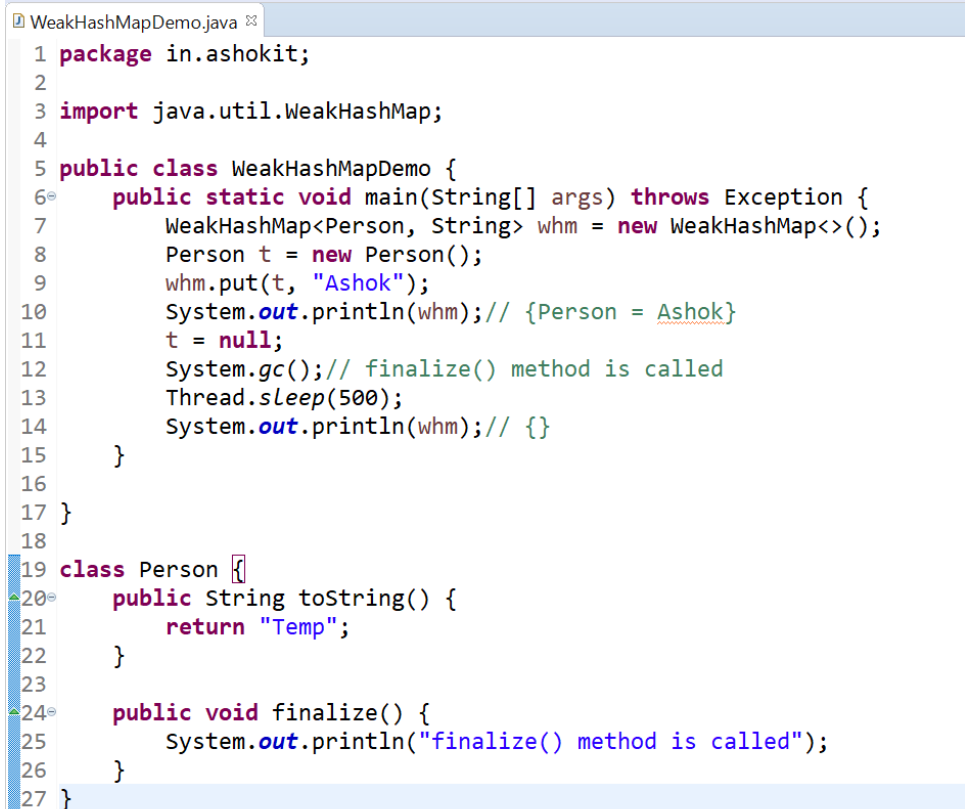
Here 10==i2 is false

WeakHashMap:

-> It is exactly same as HashMap except the following differences.

-> In the case of normal HashMap, an Object is not eligible for Garbage Collection even though it does not have any references if it is associated with HashMap. That means HashMap dominates Garbage Collector.

-> But in the case of WeakHashap if an Object does not have any references, then it is always eligible GC even though if it is associated with WeakHashMap that means GC dominates WeakHashMap.



```
1 package in.ashokit;
2
3 import java.util.WeakHashMap;
4
5 public class WeakHashMapDemo {
6     public static void main(String[] args) throws Exception {
7         WeakHashMap<Person, String> whm = new WeakHashMap<>();
8         Person t = new Person();
9         whm.put(t, "Ashok");
10        System.out.println(whm); // {Person = Ashok}
11        t = null;
12        System.gc(); // finalize() method is called
13        Thread.sleep(500);
14        System.out.println(whm); // {}
15    }
16 }
17
18
19 class Person {
20     public String toString() {
21         return "Temp";
22     }
23
24     public void finalize() {
25         System.out.println("finalize() method is called");
26     }
27 }
```

In the above program if we replace WeakHashMap with normal HashMap then Object won't be destroyed by the garbage collector.

Properties:

- 1) Properties class is the child class of Hashtable.
- 2) If anything changes frequently such type of values not recommended to hardcode in java application because for every change we have to recompile, rebuild and redeploy the application and even server restart also required. Sometimes it creates big business impact to the client.
- 3) Such type of variables we have to hardcode in property files and we have to read the values from the property files.
- 4) The main advantage in this approach is if there is any change in property files automatically those changes will be available to java application just redeployment is enough.
- 5) By using Properties Object, we can read and hold properties from property files into java application.

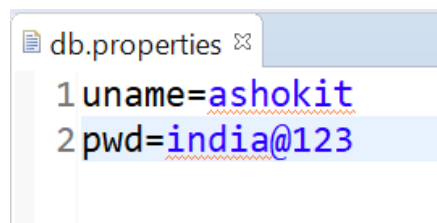
Constructor:

Properties p=new Properties ();

In Properties both key and value should be String type only.

Methods:

- 1) String getProperty (String propertyName);
- 2) String setProperty(String propertyName,String propertyValue);
- 3) Enumeration propertyNames();
- 4) void load(InputStream is);
- 5) void store(OutputStream os,String comment);



```
db.properties
1 username=ashokit
2 pwd=india@123
```

```
PropertiesDemo.java
1 package in.ashokit;
2
3 import java.io.FileInputStream;
4
5
6
7 public class PropertiesDemo {
8
9     public static void main(String[] args) throws IOException {
10         Properties p = new Properties();
11
12         FileInputStream fis = new FileInputStream("db.properties");
13         p.load(fis);
14
15         System.out.println(p); // {uname=system, pwd=manager}
16     }
17 }
18
```

Collections

- > Collections class is one of the utility classes in Java Collections Framework.
 - > The java.util package contains the Collections class.
 - > Collections class is basically used with the static methods that operate on the collections or return the collection.
 - > All the methods of this class throw the NullPointerException if the collection or object passed to the methods is null.
- Learn Here.. Lead Anywhere..!!*

Methods

- 1) public static Collection synchronizedCollection(Collection c);
- 2) public static Set synchronizedSet(Set s);
- 3) public static List synchronizedList(List list);
- 4) public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
- 5) public static SortedSet synchronizedSortedSet(SortedSet s);
- 6) public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);

Java String tokenizer

- > StringTokenizer is a class present in the java.util package and it is used to break a String into tokens based on provided delimiter.
- > Delimiter can be specified either at the time of object creation or on a per-token basis.

Following are the constructors in StringTokenizer class

1. `StringTokenizer(String str)`
2. `StringTokenizer(String str, String delim)`
3. `StringTokenizer(String str, String delim, booleanreturnValue)`

Following are the methods in StringTokenizer class

1. `booleanhasMoreTokens()`
2. `String nextToken()`
3. `String nextToken(String delim)`
4. `booleanhasMoreElements()`
5. `Object nextElement()`
6. `intcountTokens()`

// Java program to split a String using space as delimiter

```
Demo.java
1 package in.ashokit;
2
3 import java.util.StringTokenizer;
4
5 public class Demo {
6
7     public static void main(String[] args) throws Exception {
8         StringTokenizer obj = new StringTokenizer("Ashok IT Java", " ");
9         while (obj.hasMoreTokens()) {
10             System.out.println(obj.nextToken());
11         }
12     }
13 }
```

// java program to split a String using colon as delimiter

```
Demo.java
1 package in.ashokit;
2
3 import java.util.StringTokenizer;
4
5 public class Demo {
6
7     public static void main(String[] args) throws Exception {
8         String a = " : ";
9         String b = "Welcome : to : ashokit : . : How : are : You : ?";
10        StringTokenizer c = new StringTokenizer(b, a);
11        int count1 = c.countTokens();
12        for (int i = 0; i < count1; i++)
13            System.out.println("token [" + i + "] : " + c.nextToken());
14        StringTokenizer d = null;
15        while (c.hasMoreTokens()) {
16            System.out.println(d.nextToken());
17        }
18    }
19 }
```

Knowledge – Check

- 1) What is Collection and why we need it?
- 2) What is Collection framework?
- 3) What is List and when to use it?
- 4) How ArrayList works internally?
- 5) How LinkedList works internally?
- 6) When to use ArrayList and When to Use LinkedList?
- 7) What is Cursor and How many cursors available?
- 8) What is Set?
- 9) How HashSet works internally?
- 10) What is TreeSet and when to use it?
- 11) What is Comparable?
- 12) What is Comparator?
- 13) How to sort Objects?
- 14) What is Map and when to use it?
- 15) What is Hash Map?
- 16) How HashMap works internally?
- 17) How to iterate a Map?
- 18) What is Weak HashMap?
- 19) What is Identity HashMap?
- 20) What is Collections?
- 21) What is Properties Class?
- 22) What are Fail-fast collection and Fail-Safe Collection?.