

Collections

Topics Covered Under This Chapter:

- Introduction
 - Collection
 - List
 - Cursors
 - Set
 - Queue
 - Map
 - Collections
 - Arrays
 - Concurrent Collections
 - Generics
- 

1.. Introduction

What we will do if someone asked us to declare some few variables, just say three variables to hold some information of Customer, say Name, Email and Contact Number. We can simply define these variables as:

```
String name = "Vikash";  
String email = "vikash.gnss@gmail.com";  
String contact = "9079304353";
```

But what if Customer has 10,000 fields? Are we going to declare all 10,000 Variables just like above? Answer is obviously no, because if we do so, we will lose the readability of the program and it's a very worst programming practice.

We can achieve this by using Arrays concept of Java. In java, Arrays is an index-based data structure which we can use to hold 10,000 fields of Customers. This will improve the readability of the program. But there are Certain limitations of Arrays in Java:

- Arrays are Fixed in Size. We cannot change the size of an array once we have already created it. So, to use array it is compulsory that we should know its size in advance.

- Arrays hold data of homogenous type.

```
Student[] student = new Student[10000];  
student[0] = new Student();  
student[1] = new Customer();  
//Compile Time error: Incompatible Type found: Customer required: Student
```

But this problem can be solved if we declare an array of Object Type which can hold any types of data as Object class is the superclass of all other class in Java.

```
Object[] student = new Object[10000];  
student[0] = new Student();  
student[1] = new Customer();
```

- Arrays are not implemented based on some standard data structure. Hence, ready-made method support are not available in arrays. For each requirement, we have to write our own method explicitly which increases complexity of programming.

To overcome these limitations of Arrays, a new concept called **Collections** was introduced in Java.

- Collections are growable in nature i.e., size of collections can be increased or decreased based on requirements.
- Collections can store both Homogenous and Heterogenous data.
- Every collection class is implemented based on some standard data structure. Collections provides readymade methods for every requirement.

Arrays or Collections - Which to Use and When?

When memory utilization is the topmost concern then it is highly recommended to use Collections and When performance is the topmost concern then it is highly recommended to use Arrays. Also, arrays work with both primitive data types and Objects but collections work only with Objects.

Differences between Arrays and Collection

- Arrays are fixed in Size Whereas Collections are growable in nature.
- Arrays are not recommended to use While Collections are highly recommended to use when talking about Memory point of View.
- Arrays are recommended to use While Collections are not recommended to use when talking about Performance Point of View.

- Arrays can hold only homogenous data type elements while Collections can hold both Homogenous and Heterogenous elements.
- There is no underlying data structure for arrays and hence there is no support for readymade methods while Every collection is implemented based on some standard data structure and hence support of readymade method is available.
- Arrays can hold both primitive and Object types while Collections can hold only objects but not primitive types.

Collection framework provides several classes or Interfaces with the help of which we can create various kinds of Collections i.e., a group of individual objects.

Interfaces Present In Collection Framework

- Collection
- List
- Set, SortedSet, NavigableSet
- Queue
- Map, SortedMap, NavigableMap



2.. Collection

If we want to represent a group of individual objects as a single entity then we should go for Collection. Collection is an Interface which defines most common methods which are used by almost every Collection Objects. In general, Collection Interface is considered as a root interface of Collection Framework. There is no concrete class which implements Collection Interface directly.

Difference between Collection and Collections

- Collection is an Interface while Collections is a class.
- Collection defines most common methods which are used by almost every Collection Objects while Collections Class defines a various utility method for Collection Objects like sorting, searching etc.

Usually, we can use collections to hold object and also for transfer the object from One location to another. To provide support for this requirement, every collection class by default implements **Serializable** and **Cloneable** Interfaces. Some collections also implements **RandomAccess** Interface. RandomAccess Interface is present in java.util package and it has no method defined in it. It is a Marker Interface, where require ability will be provided automatically by the JVM.

Methods Present in Collection Interface

- **boolean add(Object O)**
Add an Object O to the collection
- **boolean addAll(Collection c)**
Add all the objects present in a Collection c to the Collection.
- **boolean remove(Object O)**
Removes an Object O from the Collection.
- **boolean removeAll(Collection c)**
Removes all the Object present in a Collection c from the Collection.
- **boolean retainAll(Collection c)**
Removes all other object from the Collection except the one present in Collection c.
- **void clear()**
Clears the collection i.e., remove all the objects from Collection.
- **boolean contains(Object O)**
Returns true if Object O is present in a collection else returns false.
- **boolean containsAll(Collection c)**
Returns true if a collection contains all the objects present in Collection C else returns false.
- **boolean isEmpty()**
Returns true is Collection is Empty else it returns false.
- **int size()**
Returns the Size of Collection.

- **Object[] toArray()**

Convert Collection to Array

- **Iterator iterator()**

Returns the Iterator for the collection for Traversing i.e., get the object of Collection One by One.

There is no concrete class which implements Collection Interface directly.

How to get Synchronized Version of Any Collection Object?

Collections class provides some methods which can be used to convert the non-synchronized collection to synchronized collection.

```
public static List synchronizedList(List l);
```

```
public static Set synchronizedSet(Set s);
```

```
public static Map synchronizedMap(Map m);
```

In these three methods we are going to pass the non-synchronized List, Set or Map in the argument and it will return the Synchronized List, Set or Map.

Example:

```
ArrayList al = new ArrayList();
```

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

Where al is a non-synchronized list while l is a synchronized list.

3.. List

- **List** is a Child Interface of **Collection**.
- 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.
- We can preserve Insertion order via Index values and we can also differentiate duplicate Objects with the help of Index. Hence, Index will play a very important role in List Interface and its implementation classes.
- Since List is a Child Interface of Collection, so it can use all the methods of Collection Interface. Apart from that there are several methods specific to List Interface.

Methods Present In List Interface

- **void add(int index, Object O)**
To add an Object O at specified index in a List.
- **boolean addAll(int index, Collection C)**
To add a group of Objects present in a Collection C to the List starting at specified Index.
- **Object get(int index)**
To return the Object present at specified index of the List.
- **Object remove(int index)**
To remove the Object from List present at specified Index
- **Object set(int index, Object O)**
Replace the Object present at specified index of List with the Object O and returns the Old Object.
- **int indexOf(Object O)**
Returns the First Occurrence of Object O in a List.
- **int lastIndexOf(Object O)**
Returns the Last Occurrence of Object O in a List.
- **ListIterator listIterator()**
To get the Objects present in a List One by One.

The classes which Implement List Interface are:

- ArrayList
- LinkedList
- Vector
- Stack

3.1 ArrayList

- The underlying data structure is Resizable or Growable Array.
- Duplicates Objects are allowed.
- Insertion Order is Preserved.
- Heterogenous Objects are allowed. (Except TreeSet and TreeMap everywhere Heterogenous Objects are Allowed).
- Allows null insertion.
- ArrayList implements Serializable and Clonable Interface.

- ArrayList Class RandomAccess Interface. So that any random element we can access with the same speed.
- ArrayList is the best choice when we perform retrieval operations frequently because ArrayList implements RandomAccess Interface. While, ArrayList is worst choice if our frequent operation is insertion or deletion in the middle because it requires several internal shift operations to perform

Constructor of ArrayList

- *ArrayList l = new ArrayList();*
Creates an empty ArrayList l with default initial capacity of 10. Once ArrayList reaches its max Capacity then internally a new ArrayList is created with new capacity as:
New Capacity = (Current Capacity * 3/2) + 1
- *ArrayList l = new ArrayList(int initialCapacity);*
Creates an empty ArrayList with specified initial capacity.
- *ArrayList l = new ArrayList(Collection C);*
Creates an equivalent ArrayList for the elements present in specified Collection C. C can be any collection here.

Code Example



```

Main.java
1- import java.util.*;
2 class ArrayListDemo
3 {
4     public static void main(String[] args)
5     {
6         ArrayList l = new ArrayList();
7         l.add("A");
8         l.add(10);
9         l.add("A");
10        l.add(null);
11        System.out.println(l);
12        System.out.println(l.size());
13        l.remove(1);
14        System.out.println(l);
15        l.add(2, "W");
16        l.add("N");
17        System.out.println(l);
18        System.out.println(l.indexOf("A"));
19        System.out.println(l.lastIndexOf("A"));
20    }
21 }

```

Output

```

java -cp /tmp/ryuJ2SHOMX ArrayListDemo
[A, 10, A, null]
4
[A, A, null]
[A, A, W, null, N]
0
1

```

Differences between ArrayList and Vector

- In ArrayList, No method is Synchronized while in Vector Every method is synchronized
- At a time, multiple threads are allowed to perform operation on ArrayList object, hence ArrayList object is not Thread Safe while At a time only one thread is allowed to perform operation on Vector object, hence Vector object is Thread Safe.
- Relatively performance in ArrayList is high because Threads are not required to wait to perform operation on ArrayList Object while Relatively performance in Vector is low because Threads are required to wait to perform operation on Vector Object.
- ArrayList is non legacy and introduced in 1.2 Version while Vector is legacy and introduced in 1.0 Version.

3.2 LinkedList

- The underlying data Structure for LinkedList in Java is Doubly Linked List.
- Duplicates are allowed and insertion order is preserved.
- It can also hold both homogenous and heterogenous Objects.
- Null insertion is possible.
- LinkedList implements Serializable and Clonable Interface but not RandomAccess Interface.
- LinkedList is the best choice if our frequent operation is insertion or deletion in the middle. While LinkedList is the worst choice if our frequent operation is retrieval operation.

Constructor of LinkedList


- `LinkedList l = new LinkedList();`
Creates an empty LinkedList Object l.
- `LinkedList l = new LinkedList(Collection c);`
Creates an equivalent LinkedList of the elements present in Collection c passed in the argument.

LinkedList Specific Methods

Usually, we can implement Stack and Queue by using LinkedList. To achieve this functionality, LinkedList class defines the following specific methods.

- **void addFirst(Object O)**
Add the Object O as a first element of the LinkedList.
- **void addLast(Object O)**
Add the Object O as a last element of the LinkedList.
- **Object getFirst()**
Returns the first element of the LinkedList.
- **Object getLast()**
Returns the Last element of the LinkedList.
- **Object removeFirst()**
Remove the first Object from the LinkedList and also returns it.
- **Object removeLast()**
Remove the last Object from the LinkedList and also returns it.

Code Example

Main.java	Run	Output
<pre>1 import java.util.*; 2 class LinkedListDemo 3 { 4 public static void main(String[] args) 5 { 6 LinkedList l = new LinkedList(); 7 l.add("Vikash"); 8 l.add(28); 9 l.add(null); 10 l.add("Vikash"); 11 System.out.println(l); 12 l.set(0,"Swastik"); 13 System.out.println(l); 14 l.add(0,"Softwares"); 15 System.out.println(l); 16 l.removeLast(); 17 System.out.println(l); 18 l.addFirst("Hi!"); 19 System.out.println(l); 20 } 21 }</pre>		<pre>java -cp /tmp/CaXOGLP1TB LinkedListDemo [Vikash, 28, null, Vikash] [Swastik, 28, null, Vikash] [Softwares, Swastik, 28, null, Vikash] [Softwares, Swastik, 28, null] [Hi!, Softwares, Swastik, 28, null]</pre>

Differences between ArrayList and LinkedList

- ArrayList is the best choice when we perform retrieval operations frequently because ArrayList implements RandomAccess Interface while LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
- ArrayList is worst choice if our frequent operation is insertion or deletion in the middle because it requires several internal shift operations to perform while LinkedList is the worst choice if our frequent operation is retrieval operation.
- In ArrayList, objects are stored in consecutive memory locations as ArrayList uses Growable Array as the underlying data structure while In LinkedList, objects are not stored in Consecutive memory locations as LinkedList use Doubly Linked List as Underlying Data Structure.

3.3 Vector

- The underlying data structure for Vector is Resizable or Growable Array.
- It can store both homogenous and heterogenous elements, duplicates and also preserves insertion order.
- It allows null insertion.
- It implements Serializable, Cloneable and RandomAccess Interface.
- All methods in Vector class are synchronized, hence it is Thread Safe.

Constructor of Vector

- *Vector v = new Vector();*
Creates an empty vector v with default initial capacity of 10. Once Vector reaches its max Capacity then internally a new Vector is created with new capacity as:
New Capacity = Current Capacity * 2
- *Vector v = new Vector(int initialCapacity);*
Creates an empty Vector v with specified initial capacity.
- *Vector v = new Vector(Collection C);*
Creates an equivalent Vector for the elements present in Collection C specified in the argument.
- *Vector v = new Vector(int initialCapacity, int incrementValue);*
Creates an empty Vector V with specified initial size and once the vector reaches to its size, it will create a vector with new capacity as:
New Capacity = Current Capacity + incrementValue;

Vector Specific Methods

- **addElement(Object O)**
Add an Object O to the Vector. It is equivalent to add(Object O) of the Collection Interface and add(int index, Object O) of List Interface.
- **removeElement(Object O)**
Remove an Object O from the Vector. It is equivalent to remove(Object O) of Collection Interface
- **removeElementAt(int index)**
Remove an object present at specified index of Vector. It is equivalent to remove(int index) of List Interface.

- **removeAllElements()**

It will remove all the objects from the Vector. It is equivalent to clear() method of Collection Interface.

- **Object elementAt(int index)**

Returns the object present at specified index of Vector. It is equivalent to get(int index) method of List Interface.

- **Object firstElement()**

Returns the First element of Vector.

- **Object lastElement()**

Returns the Last element of Vector.

- **int capacity()**

Returns the Capacity of Vector.

Code Example

Main.java	Run	Output
<pre>1- import java.util.*; 2 class VectorDemo 3 { 4 public static void main(String[] args) 5 { 6 Vector vector = new Vector(); 7 System.out.println("Default Initial Capacity " + vector.capacity()); 8 for(int i=1; i<=10;i++) 9 { 10 vector.add(i); 11 } 12 System.out.println("After Adding 10 Elements " + vector.capacity()); 13 vector.addElement("Vikash"); 14 System.out.println("After Adding 1 Elements Beyond Capacity " + vector.capacity()); 15 vector.addElement("Vikash"); 16 System.out.println(vector); 17 vector.removeElement(2); 18 System.out.println(vector); 19 System.out.println(vector.firstElement()); 20 System.out.println(vector.lastElement()); 21 } 22 }</pre>		<pre>java -cp /tmp/zVc53tseHS VectorDemo Default Initial Capacity 10 After Adding 10 Elements 10 After Adding 1 Elements Beyond Capacity 20 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, Vikash, Vikash] [1, 3, 4, 5, 6, 7, 8, 9, 10, Vikash, Vikash] 1 Vikash</pre>

What if we change the incremental value to 5 instead of making the capacity double. We can do that so by making following change in a code:

Main.java	Run	Output
<pre>1- import java.util.*; 2 class VectorDemo 3 { 4 public static void main(String[] args) 5 { 6 Vector vector = new Vector(10,5); 7 System.out.println("Default Initial Capacity " + vector.capacity()); 8 for(int i=1; i<=10;i++) 9 { 10 vector.add(i); 11 } 12 System.out.println("After Adding 10 Elements " + vector.capacity()); 13 vector.addElement("Vikash"); 14 System.out.println("After Adding 1 Elements Beyond Capacity " + vector.capacity()); 15 vector.addElement("Vikash"); 16 System.out.println(vector); 17 vector.removeElement(2); 18 System.out.println(vector); 19 System.out.println(vector.firstElement()); 20 System.out.println(vector.lastElement()); 21 } 22 }</pre>		<pre>java -cp /tmp/zVc53tseHS VectorDemo Default Initial Capacity 10After Adding 10 Elements 10 After Adding 1 Elements Beyond Capacity 15 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, Vikash, Vikash] [1, 3, 4, 5, 6, 7, 8, 9, 10, Vikash, Vikash] 1 Vikash</pre>

3.4 Stack

- Stack is a Child Class of Vector Class.
- Stack is specially designed to have the Last In First Out (LIFO) functionality.

Constructor of Stack

- `Stack stack = new Stack();`

Stack Specific Methods

- **Object push(Object O)**
Push or Add an Object O to the Stack.
- **Object pop()**
Remove and Return the Element present at the top of the Stack.
- **Object peek()**
Return the element present at the top of the stack without removing it.
- **boolean empty()**
Returns true if stack is empty else returns false.
- **int search(Object O)**
Returns the Offset of the element from the stack. If element is not present in a Stack it returns -1. Offset means the position of element with respect to top of the Stack.

Code Example

Main.java	Run	Output
<pre>1 import java.util.*; 2 class StackDemo 3 { 4 public static void main(String[] args) 5 { 6 Stack stack = new Stack(); 7 System.out.println(stack.empty()); 8 stack.push("A"); 9 stack.push("B"); 10 stack.push("C"); 11 stack.push("D"); 12 System.out.println(stack.empty()); 13 System.out.println(stack); 14 stack.pop(); 15 System.out.println(stack); 16 stack.pop(); 17 System.out.println(stack); 18 System.out.println(stack.peek()); 19 System.out.println(stack.search("A")); 20 System.out.println(stack.search("B")); 21 System.out.println(stack.search("Z")); 22 } 23 }</pre>		<pre>java -cp /tmp/zVcS3tseHS StackDemo true false [A, B, C, D] [A, B, C] [A, B] B 2 1 -1</pre>

4.. Cursors

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

There are three types of Cursors in Java:

1. Enumeration
2. Iterator
3. ListIterator

4.1 Enumeration

- We can use Enumeration to get Objects one by one from Legacy Collection Objects.
- We can create Enumeration Object by calling elements() methods of Vector Class.

public Enumeration elements();

Example: *Enumeration e = v.elements();*

Methods Present in Enumeration Class

- **boolean hasMoreElements()**
Returns true until Enumeration has elements in it else return false.
- **Object nextElement()**
Returns the next Element present in an Enumeration

Code Example

```
Main.java
1- import java.util.*;
2 class EnumerationDemo
3- {
4     public static void main(String[] args)
5     {
6         Vector v = new Vector();
7         for(int i=1; i<=10; i++)
8         {
9             v.addElement(i);
10        }
11        System.out.println(v);
12        Enumeration e = v.elements();
13        while(e.hasMoreElements())
14        {
15            Integer i = (Integer) e.nextElement();
16            if(i % 2 == 0){
17                System.out.println(i);
18            }
19        }
20    }
21 }
```

Output

```
java -cp /tmp/ZI6PB5kTY1 EnumerationDemo
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
4
6
8
10
```

Limitations of Enumeration Cursor

- Enumeration is applicable for only Legacy classes like Vector and Stack. It is not a Universal Cursor and we cannot apply it for ArrayList or LinkedList.
- By using Enumeration we can get only read access but can't perform remove operation.
- To overcome above limitations we should go for **Iterator** Cursor.

4.2 Iterator

- We can apply Iterator Concept for any Collection Object. Hence it is a Universal Cursor.
- By using Iterator we can perform both read and remove operations.
- We can create Iterator Object by calling iterator() method of Collection Interface.

public Iterator iterator();

Example: *Iterator it = c.iterator();* Here c is an object of any Collection.

Methods Present in Iterator Class

- **boolean hasNext()**
Returns true until Iterator has next elements in it else return false.
- **Object next()**
Returns the next Object present in an Iterator.
- **void remove()**
Removes the current object from the Iterator.

Code Example

Main.java	Output
<pre>1- import java.util.*; 2 class EnumerationDemo 3- { 4 public static void main(String[] args) 5 { 6 ArrayList a = new ArrayList(); 7 for(int i=1; i<=10; i++) 8 { 9 a.add(i); 10 } 11 System.out.println(a); 12 Iterator it = a.iterator(); 13 while(it.hasNext()) 14 { 15 Integer i = (Integer) it.next(); 16 if(i % 2 == 0){ 17 System.out.println(i); 18 it.remove(); 19 } 20 } 21 System.out.println(a); 22 } 23 }</pre>	<pre>java -cp /tmp/ZI6P8SkTY1 EnumerationDemo [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 2 4 6 8 10 [1, 3, 5, 7, 9]</pre>

Limitations of Iterator Cursor

- Enumeration and Iterator both are single direction cursor but not bi-directional cursor. They can move only in forward direction but not in backward direction.
- With the help of Iterator we can only perform read and remove operation. We cannot add any object or replace any existing object using Iterator.
- To overcome these Limitations, we should go for **ListIterator**.

4.3 ListIterator

- ListIterator is a bi-directional cursor which can move both in forward and backward direction.
- With the help of ListIterator, apart from read and remove operation we can also perform add and replace operation.
- We can create ListIterator Object by calling listIterator() method of List Interface.
public ListIterator listIterator();

Example: *ListIterator lit = l.listIterator();* Here l is an object of any List.

Methods Present in ListIterator Class

ListIterator is a Child Interface of Iterator. Hence, all methods present in Iterator Interface are by default present in ListIterator as well. Apart from those methods there are some specific methods contained by ListIterator.

- **boolean hasNext()**
Returns true until next element is present in List object else return false.
- **Object next()**
Return the next element with respect to current element in a list
- **int nextIndex()**
Return the index of next element of a List
- **boolean hasPrevious()**
Returns true until previous element is present in List object else return false.
- **Object previous()**
Return the previous element with respect to current element in a list.
- **int previousIndex()**
Return the index of previous element of a List.
- **void remove()**
Remove the current element from the List.
- **void add(Object O)**
Add a new Object O next to the current element in a List.
- **void set(Object O)**
Replace the current Object with Object O.

Note: The most powerful cursor is ListIterator. But its limitations is it can only be used with List Objects. It is not a universal Iterator.

Code Example

Main.java	Output
<pre> 1- import java.util.*; 2 class EnumerationDemo 3 { 4 public static void main(String[] args) 5 { 6 ArrayList a = new ArrayList(); 7 for(int i=1; i<=10; i++) 8 { 9 a.add(i); 10 } 11 System.out.println(a); 12 ListIterator it = a.listIterator(); 13 while(it.hasNext()) 14 { 15 Integer i = (Integer) it.next(); 16 if(i % 2 == 0){ 17 System.out.println(i); 18 it.remove(); 19 } 20 else 21 { 22 it.set(i+5); 23 } 24 } 25 System.out.println(a); 26 } 27 }</pre>	<pre> java -cp /tmp/ZI6PB5kTY1 EnumerationDemo [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 2 4 6 8 10 [6, 8, 10, 12, 14]</pre>

Main.java	Run	Output
<pre>1- import java.util.*; 2 class EnumerationDemo 3 { 4 public static void main(String[] args) 5 { 6 Vector v = new Vector(); 7 Enumeration e = v.elements(); 8 Iterator itr = v.iterator(); 9 ListIterator litr = v.listIterator(); 10 System.out.println(e.getClass().getName()); 11 System.out.println(itr.getClass().getName()); 12 System.out.println(litr.getClass().getName()); 13 } 14 }</pre>	 	<pre>java -cp /tmp/ZI6P8SkTYI EnumerationDemo java.util.Vector\$1 java.util.Vector\$Itr java.util.Vector\$listItr</pre>



5.. Set

Set is a Child Interface of Collection. If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order is not need of to be preserved then we should go for Set Interface.

There are some classes which implements Set Interface:



HashSet
LinkedHashSet

Set Interface doesn't contain any new methods. Only Collection Interface methods we have to use.

5.1 HashSet

- The underlying data structure for HashSet is Hashtable.
- Duplicates objects are not allowed.
- All the objects in a HashSet are inserted based on Hash Code. This is the reason why insertion Order is not preserved in HashSet
- Null Insertion is Possible but since duplicates are not allowed in HashSet, we can add null only once.
- HashSet can store both heterogenous and Homogenous Objects.
- HashSet implements Serializable and Clonable Interface but doesn't implement RandomAccess Interface.
- Whenever our frequent operation is search operation, then we should always go for HashSet.

Note: In HashSet, duplicates are not allowed. If we are trying to add duplicates then we won't get any compile time or run time errors and add() method simply returns false.

Main.java			Run	Output
<pre>1- import java.util.*; 2 class HashSetDemo 3- { 4 public static void main(String[] args) 5 { 6 HashSet h = new HashSet(); 7 System.out.println(h.add("Vikash")); 8 System.out.println(h.add("Vikash")); 9 } 10 }</pre>				<pre>java -cp /tmp/ZI6PBskTY1 HashSetDemo true false</pre>

Constructor of HashMap

- *HashSet hashSet = new HashSet();*
Creates an empty HashSet with default initial capacity of 16 and default fill ratio as 0.75.
- *HashSet hashSet = new HashSet(int initialCapacity);*
Creates an empty HashSet with specified initial capacity and fill ratio will be default i.e., 0.75.
- *HashSet hashSet = new HashSet(int initialCapacity, float fillRatio);*
Creates an empty HashSet with specified initial capacity and specified fillRatio.
- *HashSet hashSet = new HashSet(Collection C);*
Creates an equivalent HashSet of specified Collection C.

Fill Ratio or **Load Factor** is the ratio or factor which determines after filling how much ratio a new HashSet will be created. For example, fill ratio 0.75 means after filling 75% ratio of HashSet, a new HashSet will be created Internally.

Code Example

<pre>Main.java 1- import java.util.*; 2 class HashSetDemo 3 { 4 public static void main(String[] args) 5 { 6 HashSet h = new HashSet(); 7 h.add("B"); 8 h.add("C"); 9 h.add("D"); 10 h.add("Z"); 11 h.add(null); 12 h.add(10); 13 h.add("B"); 14 15 System.out.println(h.add("Z")); 16 System.out.println(h); 17 } 18 }</pre>	<pre>Output java -cp /tmp/ZI6PBSkTY1 HashSetDemo false [null, B, C, D, Z, 10]</pre>
--	---

5.2 LinkedHashMap

- LinkedHashMap is a Child Class of HashSet.
- The underlying data structure for LinkedHashMap is Linked List and Hash Table.
- Since it uses Linked List and Hash Table both, Insertion Order is Preserved.
- Except this above functionality, it is similar to HashSet.

Code Example

<pre>Main.java 1- import java.util.*; 2 class LinkedHashMapDemo 3 { 4 public static void main(String[] args) 5 { 6 LinkedHashMap h = new LinkedHashMap(); 7 h.add("B"); 8 h.add("C"); 9 h.add("D"); 10 h.add("Z"); 11 h.add(null); 12 h.add(10); 13 h.add("B"); 14 15 System.out.println(h.add("Z")); 16 System.out.println(h); 17 } 18 }</pre>	<pre>Output java -cp /tmp/ZI6PBSkTY1 LinkedHashMapDemo false [B, C, D, Z, null, 10]</pre>
--	---

In general, we can use LinkedHashMap to develop Cache based applications where duplicates are not allowed and insertion order preserved

5.3 SortedSet

- **SortedSet** is a Child of **Set** Interface.
- If we want to represent a group of individual objects as a single entity where duplicates are not allowed but all objects are inserted according to some sorting order then we should go for SortedSet Interface.
- There is no class which implements SortedSet but we have a Child of SortedSet for that we have an implementation class.

Methods Defined by SortedSet Interface

SortedSet is a Child of Set Interface which is a child of Collection Interface. Set Interface doesn't have any methods of its own. It inherits all the methods of Collection Interface. But SortedSet Interface apart from inheriting all the methods of Collection Interface, defines some methods of its own. The following methods are defined by SortedSet.

- **Object first()**
Returns the First Object of SortedSet
- **Object last()**
Returns the Last Object of SortedSet
- **SortedSet headSet(Object O)**
Return the SortedSet whose elements are $< O$
- **SortedSet tailSet(Object O)**
Return the SortedSet whose elements are \geq Object O
- **SortedSet subSet(Object start, Object end)**
Return the SortedSet whose elements are \geq Object start and $<$ Object end.
- **Comparator comparator()**
Returns Comparator Object that describes underlying sorting technique. If we are using default natural sorting order then we will get null. For numbers default sorting order is ascending order while for string it is Alphabetical Order.

5.4 TreeSet

- TreeSet is the implementation class of NavigableSet which is a child Interface of SortedSet. So, TreeSet is the implementation class of both SortedSet and NavigableSet.
- The underlying data structure for TreeSet is Balanced Tree also known as B-Tree.
- No Duplicates are allowed and insertion order is also not preserved.
- Allows null insertion only for one time that too only until version 1.6 and also Heterogenous Objects are not allowed.
- TreeSet implements Serializable and Clonable Interface but doesn't implement RandomAccess Interface.
- All Elements are inserted as per some sorting Order. It can be default natural sorting order or any customized sorting order.

Constructors of TreeSet

- *TreeSet t = new TreeSet();*
Creates an empty TreeSet t with default natural sorting order.
- *TreeSet t = new TreeSet(Comparator C);*
Creates an empty TreeSet t with Customized Sorting Order. The Customized Sorting order can be implemented by using Comparator Object C.
- *TreeSet t = new TreeSet(Collection C);*
Creates an equivalent TreeSet of specified Collection C with default natural sorting Order.

- `TreeSet t = new TreeSet(SortedSet S);`
Creates a `TreeSet` equivalent to `SortedSet S` with Same sorting order which `S` carries.

Code Example

Main.java	Output
<pre> 1- import java.util.*; 2 class TreeSetDemo 3 { 4 public static void main(String[] args) 5 { 6 TreeSet h = new TreeSet(); 7 h.add(100); 8 h.add(101); 9 h.add(104); 10 h.add(106); 11 h.add(110); 12 h.add(115); 13 h.add(120); 14 15 System.out.println(h.first()); 16 System.out.println(h.last()); 17 System.out.println(h.headSet(106)); 18 System.out.println(h.tailSet(106)); 19 System.out.println(h.subSet(101,115)); 20 } 21 }</pre>	<pre> java -cp /tmp/ZI6PBskTY1 TreeSetDemo 100 120 [100, 101, 104] [106, 110, 115, 120] [101, 104, 106, 110]</pre>

Code Example For TreeSet Sorting Order

Main.java	Output
<pre> 1- import java.util.*; 2 class TreeSetDemo 3 { 4 public static void main(String[] args) 5 { 6 TreeSet h = new TreeSet(); 7 h.add(100); 8 h.add(101); 9 h.add(104); 10 h.add(106); 11 h.add(110); 12 h.add(115); 13 h.add(120); 14 System.out.println(h); 15 16 TreeSet h1 = new TreeSet(); 17 h1.add("A"); 18 h1.add("a"); 19 h1.add("B"); 20 h1.add("b"); 21 h1.add("C"); 22 System.out.println(h1); 23 h1.add(1); // Throws ClassCastException 24 } 25 }</pre>	<pre> java -cp /tmp/ZI6PBskTY1 TreeSetDemo [100, 101, 104, 106, 110, 115, 120] [A, B, a, b, C] Exception in thread "main" java.lang.ClassCastException: class java.lang.String cannot be cast to class java .lang.Integer (java.lang.String and java.lang.Integer are in module java.base of loader 'bootstrap') at java.base/java.lang.Integer.compareTo(Integer.java:59) at java.base/java.util.TreeMap.put(TreeMap.java:566) at java.base/java.util.TreeSet.add(TreeSet.java:255) at TreeSetDemo.main(TreeSetDemo.java:23)</pre>

Issue with Null Insertion In TreeSet

- For Non-empty `TreeSet` if we are trying to add null then we are going to get `NullPointerException`.
- For Empty `TreeSet` as a first element, null is allowed to insert until Java version 1.6. But after inserting null if we are trying to insert any other Object then we will get `NullPointerException`. But after java version 1.7 Null is not allowed to enter into `TreeSet` as a first element also.

Main.java	Output
<pre> 1- import java.util.*; 2 class TreeSetDemo 3 { 4 public static void main(String[] args) 5 { 6 TreeSet h = new TreeSet(); 7 h.add(null); 8 } 9 }</pre>	<pre> java -cp /tmp/ZI6PBskTY1 TreeSetDemo Exception in thread "main" java.lang.NullPointerException at java.base/java.util.TreeMap.compare(TreeMap.java:1291) at java.base/java.util.TreeMap.put(TreeMap.java:536) at java.base/java.util.TreeSet.add(TreeSet.java:255) at TreeSetDemo.main(TreeSetDemo.java:7)</pre>

Comparable Interface

- Comparable interface is present in java.lang package.
- Comparable interface contains only one method which is compareTo() method.
public int compareTo(Object O);

Example – *Obj1.compareTo(Obj2);*

The compareTo() method returns:

Negative Value

If and only if Obj1 comes before Obj2

Positive Value

If and only if Obj1 comes after Obj2

0

If and only if Obj1 and Obj2 are equal.

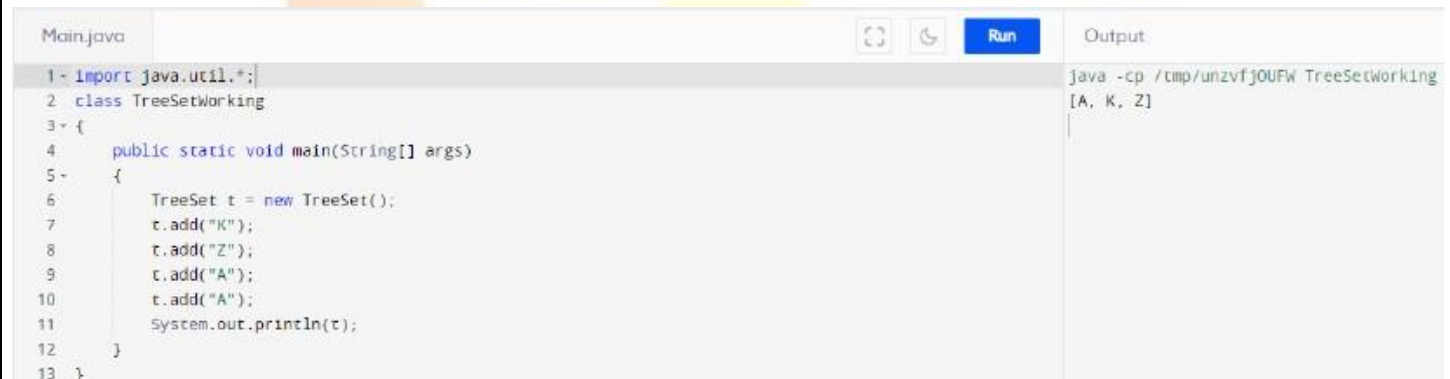


```
1 class ComparableDemo
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("A".compareTo("Z"));
6         System.out.println("A".compareTo("A"));
7         System.out.println("A".compareTo("A"));
8         System.out.println("A".compareTo(null));
9     }
10 }
```

Output

```
java -cp /tmp/unzvfj0UFW ComparableDemo
-25
10
0
Exception in thread "main" java.lang.NullPointerException
at java.base/java.lang.String.compareTo(String.java:1196)at ComparableDemo.main(ComparableDemo.java:8)
```

Internal Working of TreeSet



```
1 import java.util.*;
2 class TreeSetWorking
3 {
4     public static void main(String[] args)
5     {
6         TreeSet t = new TreeSet();
7         t.add("K");
8         t.add("Z");
9         t.add("A");
10        t.add("A");
11        System.out.println(t);
12    }
13 }
```

Output

```
java -cp /tmp/unzvfj0UFW TreeSetWorking
[A, K, Z]
```

Here we have created a TreeSet object which is depending on default natural sorting order. When we depend on default natural sorting order then while adding an element into the TreeSet, JVM will internally call compareTo() method of Comparable Interface in the following format:

Obj1.compareTo(Obj2);

Where Obj1 is the element which we are going to insert in the TreeSet and Obj2 is a root element. Root element is the first element which got inserted in the TreeSet.

- Now, when t.add("K") is called, since TreeSet is empty and there is nothing to compare, "K" gets added to the TreeSet.
- Next we are trying to add "Z" so JVM will call internally "Z".compareTo("K") which will return positive as "Z" comes after "K" so "Z" will be added right to "K".
- Now, again when we are trying to add "A" to TreeSet, again JVM will call internally "A".compareTo("K") which will return negative as "A" comes before "K" so "A" will be inserted to the left of "K".
- Again when we are trying to insert "A" again JVM will call "A".compareTo("K") which will again return negative and "A" should be added to the left of "K" but to the left of "K" we

already have one element which is "A" so again JVM will call "A".compareTo("A") which will return 0 and it implies "A" is already there in TreeSet and it shouldn't be added again.

If default natural sorting order is not available or we are not happy with default natural sorting order then we can create a customized sorting order. This we can achieve by **Comparator** Object.

Comparator Interface

- Comparator Interface is present in java.util package.
- Comparator Interface contains two methods:
public int compare(Object obj1, Object obj2);
public boolean equals(Object obj);
- The compare() method returns:
Negative Value : If and only if Obj1 comes before Obj2
Positive Value : If and only if Obj1 comes after Obj2
0 : If and only if Obj1 and Obj2 are equal.
- Whenever any class implements Comparator Interface then it is compulsory to implement compare() method but it is not compulsory to implement equals() method. This is because every class is a child of Object class and Object class has already defined the implementation of equals() method so if we don't implement equals() method in our class then it will use Object class equals() method.

```
Main.java
4      Integer I1 = (Integer) obj1;
5      Integer I2 = (Integer) obj2;
6+     if(I1 < I2){
7         return 1;
8-     }else if(I1 > I2){
9         return -1;
10-    }else{
11        return 0;
12    }
13 }
14 }
15+ class TreeSetSorting{
16-     public static void main(String[] args){
17         TreeSet t1 = new TreeSet();
18         t1.add(10); //Since TreeSet Empty 10 Added as Root
19         t1.add(0); //Calls 0.compareTo(10) returns -ve;
20         t1.add(15); //Calls 15.compareTo(10) returns +ve;
21         t1.add(5); //Calls 5.compareTo(10); returns -ve and 5.compareTo(0) return +ve;
22         t1.add(20); //Calls 20.compareTo(10) returns +ve and 20.compareTo(15) returns +ve;
23         t1.add(20); //Calls 20.compareTo(10); returns -ve and 20.compareTo(15) returns -ve
24                     //Above line also Calls 20.compareTo(20) and returns 0;
25         System.out.println(t1); //Print InOrder Traversal of Tree
26
27         TreeSet t2 = new TreeSet(new MyComparator());
28         t2.add(10); //Since TreeSet Empty 10 Added as Root
29         t2.add(0); //Calls compare(0,10);
30         t2.add(15); //Calls compare(15,10);
31         t2.add(5); //Calls compare(5,10); compare(5,0);
32         t2.add(20); //Calls compare(20,10); compare(20,15);
33         t2.add(20); //Calls compare(20,10); compare(20,15); compare(20,20);
34         System.out.println(t2); //Print InOrder Traversal of Tree
35     }
36 }
```

```
Output
java -cp /tmp/1DB8TSvFZW TreeSetSorting
[0, 5, 10, 15, 20]
[20, 15, 10, 5, 0]
```

We can also achieve default natural sorting order by using Comparator Object by modifying compare() method of Comparator Interface and return Obj1.compareTo(Obj2) from the method.

Main.java	Run	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3- public int compare(Object obj1, Object obj2){ 4- Integer I1 = (Integer) obj1; 5- Integer I2 = (Integer) obj2; 6- return I1.compareTo(I2); 7- } 8- } 9- class TreeSetSorting{ 10- public static void main(String[] args){ 11- TreeSet t2 = new TreeSet(new MyComparator()); 12- t2.add(10); 13- t2.add(0); 14- t2.add(15); 15- t2.add(5); 16- t2.add(20); 17- t2.add(20); 18- System.out.println(t2); 19- } 20- }</pre>		<pre>java -cp /tmp/IDB8TSvFZW TreeSetSorting [0, 5, 10, 15, 20]</pre>

Also,

Main.java	Run	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3- public int compare(Object obj1, Object obj2){ 4- Integer I1 = (Integer) obj1; 5- Integer I2 = (Integer) obj2; 6- return -I2.compareTo(I1); 7- } 8- } 9- class TreeSetSorting{ 10- public static void main(String[] args){ 11- TreeSet t2 = new TreeSet(new MyComparator()); 12- t2.add(10); 13- t2.add(0); 14- t2.add(15); 15- t2.add(5); 16- t2.add(20); 17- t2.add(20); 18- System.out.println(t2); 19- } 20- }</pre>		<pre>java -cp /tmp/IDB8TSvFZW TreeSetSorting [0, 5, 10, 15, 20]</pre>

We can change this ascending sorting order to descending sorting order by making a small change.

Main.java	Run	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3- public int compare(Object obj1, Object obj2){ 4- Integer I1 = (Integer) obj1; 5- Integer I2 = (Integer) obj2; 6- return -I1.compareTo(I2); 7- } 8- } 9- class TreeSetSorting{ 10- public static void main(String[] args){ 11- TreeSet t2 = new TreeSet(new MyComparator()); 12- t2.add(10); 13- t2.add(0); 14- t2.add(15); 15- t2.add(5); 16- t2.add(20); 17- t2.add(20); 18- System.out.println(t2); 19- } 20- }</pre>		<pre>java -cp /tmp/IDB8TSvFZW TreeSetSorting [20, 15, 10, 5, 0]</pre>

Also,

Main.java	Run	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3- public int compare(Object obj1, Object obj2){ 4- Integer I1 = (Integer) obj1; 5- Integer I2 = (Integer) obj2; 6- return I2.compareTo(I1); 7- } 8- } 9- class TreeSetSorting{ 10- public static void main(String[] args){ 11- TreeSet t2 = new TreeSet(new MyComparator()); 12- t2.add(10); 13- t2.add(0); 14- t2.add(15); 15- t2.add(5); 16- t2.add(20); 17- t2.add(20); 18- System.out.println(t2); 19- } 20- }</pre>		<pre>java -cp /tmp/IDB8TSvFZW TreeSetSorting [20, 15, 10, 5, 0]</pre>

We can make our TreeSet to preserve insertion order and also hold duplicates by returning a positive value from the compare() method and we can print the reverse insertion order by returning negative value from the compare() method. If we return 0 from the compare() method then TreeSet will allow only first element to get inserted and all other elements will be treated as Duplicates and will be discarded.

Main.java	Run	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3- public int compare(Object obj1, Object obj2){ 4- return 1; 5- } 6- } 7- class TreeSetSorting{ 8- public static void main(String[] args){ 9- TreeSet t2 = new TreeSet(new MyComparator()); 10- t2.add(10); 11- t2.add(0); 12- t2.add(15); 13- t2.add(5); 14- t2.add(20); 15- t2.add(20); 16- System.out.println(t2); 17- } 18- }</pre>		<pre>java -cp /tmp/IDB8TSvFZW TreeSetSorting [10, 0, 15, 5, 20, 20]</pre>

Main.java	Run	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3- public int compare(Object obj1, Object obj2){ 4- return -1; 5- } 6- } 7- class TreeSetSorting{ 8- public static void main(String[] args){ 9- TreeSet t2 = new TreeSet(new MyComparator()); 10- t2.add(10); 11- t2.add(0); 12- t2.add(15); 13- t2.add(5); 14- t2.add(20); 15- t2.add(20); 16- System.out.println(t2); 17- } 18- }</pre>		<pre>java -cp /tmp/IDB8TSvFZW TreeSetSorting [20, 20, 5, 15, 0, 10]</pre>

Main.java		Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3- public int compare(Object obj1, Object obj2){ 4- return 0; 5- } 6- } 7- class TreeSetSorting{ 8- public static void main(String[] args){ 9- TreeSet t2 = new TreeSet(new MyComparator()); 10- t2.add(10); 11- t2.add(0); 12- t2.add(15); 13- t2.add(5); 14- t2.add(20); 15- t2.add(20); 16- System.out.println(t2); 17- } 18- }</pre>		<pre>java -cp /tmp/IDB8TSvFZW TreeSetSorting [10]</pre>

Same Logic can be applied while dealing with String Objects.

5.5 NavigableSet

- NavigableSet Interface was Introduced in Java 1.6 Onwards.
- NavigableSet is a Child of SortedSet Interface. It contains several methods for Navigation Purposes.
- There is only one implementation class for NavigableSet which is TreeSet.

Methods Introduced by NavigableSet

- **Object floor(e)**
Returns highest element which is $\leq e$
- **Object lower(e)**
Returns highest element which is $< e$
- **Object ceiling(e)**
Returns Lowest element which is $\geq e$
- **Object higher(e)**
Returns Lowest element which is $> e$
- **Object pollFirst()**
Remove and Return first element
- **Object pollLast()**
Remove and Return last element.
- **NavigableSet descendingSet()**
Return NavigableSet in reverse Order.

Code Example

Main.java

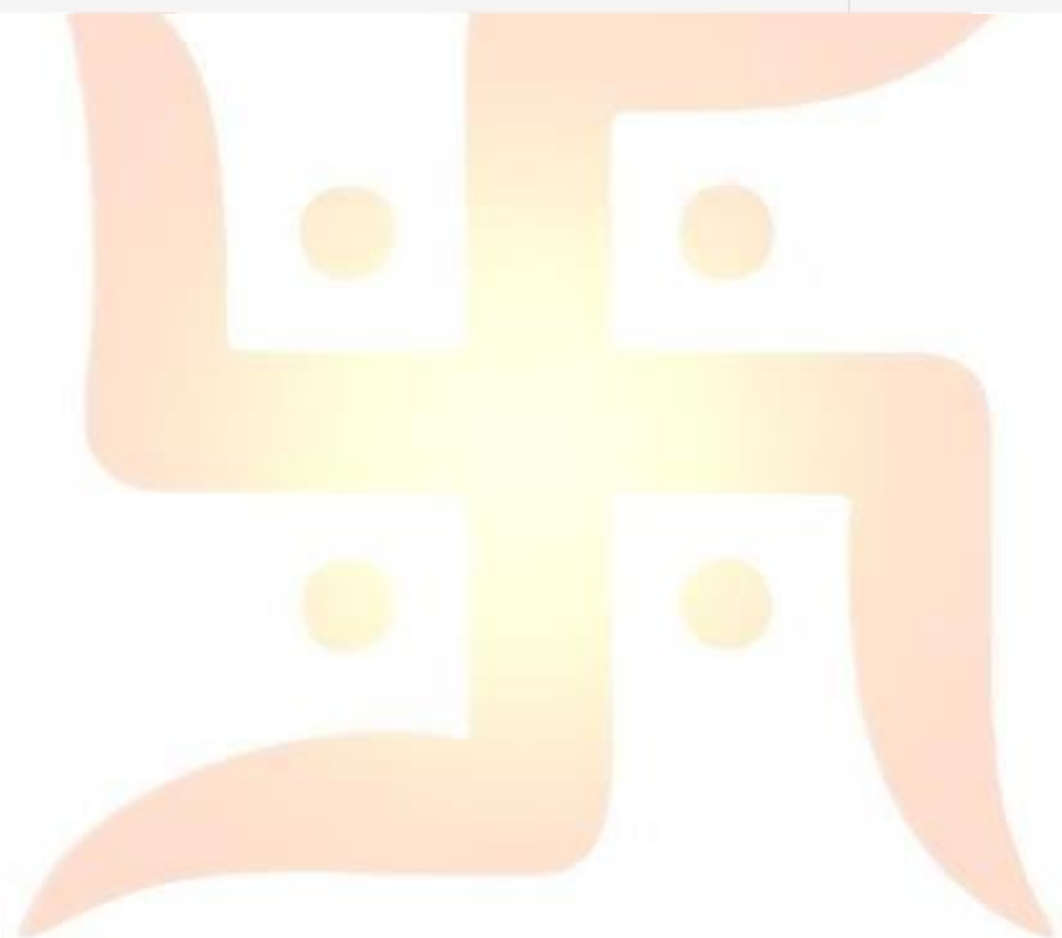


Run

Output

```
1 import java.util.*;
2 class NavigableSetDemo
3 {
4     public static void main(String[] args)
5     {
6         TreeSet<Integer> t = new TreeSet<>();
7         t.add(1000);
8         t.add(2000);
9         t.add(3000);
10        t.add(4000);
11        t.add(5000);
12        System.out.println(t);
13        System.out.println(t.ceiling(2000));
14        System.out.println(t.higher(2000));
15        System.out.println(t.floor(3000));
16        System.out.println(t.lower(2000));
17        System.out.println(t.pollFirst());
18        System.out.println(t.pollLast());
19        System.out.println(t.descendingSet());
20        System.out.println(t);
21    }
22 }
```

```
java -cp /tmp/S06QY1xbbg NavigableSetDemo
[1000, 2000, 3000, 4000, 5000]
2000
3000
3000
1000
1000
5000
[4000, 3000, 2000][2000, 3000, 4000]
```



6.. Queue

- Queue is a Child Interface of Collection.
- If we want to represent a group of individual objects prior to processing then we should go for Queue.
- Usually, Queue follows FIFO (First in First Out) order but based on our requirement, we can implement our own priority order also. For example, before sending a mail we have to store all the mail Ids in some data structure. The order in which we entered Mail Id is the same order in which Mail is sent. For this requirement Queue is the best choice.
- There are some classes which implements Queue Interface:
 1. PriorityQueue
 2. BlockingQueue which further got extended by PriorityBlockingQueue and LinkedBlockingQueue.
 3. Dequeue
- We will discuss only PriorityQueue.

Queue Specific Methods

Though Queue extends Collection Interface, so Collection Method we can use here happily but additionally, Queue has defined some specific methods which are as follows:

- **boolean offer(Object O)**
This Methods add an Object O to the Queue.
- **Object poll()**
Remove and Return the Head Element of the Queue. Head means the element in a Queue which is going to get the service first. If Queue is empty then we will get null.
- **Object remove()**
Remove and Return the Head Element of the Queue. Head means the element in a Queue which is going to get the service first. If Queue is empty then we will get NoSuchElementException.
- **Object peek()**
Return the Head Element of the Queue. Head means the element in a Queue which is going to get the service first. If Queue is empty then we will get null.
- **Object element()**
Return the Head Element of the Queue. Head means the element in a Queue which is going to get the service first. If Queue is empty then we will get NoSuchElementException.

6.1 PriorityQueue

- If we want to represent a group of individual objects prior to processing according to priority then we should go for PriorityQueue.
- The priority can be default natural sorting order or Customized Sorting Order.
- Insertion order is not preserved.
- Duplicates are not allowed.
- If we are depending on default natural sorting order then Heterogenous Objects are not allowed, while if we define a customize sorting order then both Homogenous and Heterogenous Objects are allowed.
- Null insertion is not possible in priority Queue.

Constructors of PriorityQueue

- *PriorityQueue p = new PriorityQueue();*
Creates an Empty PriorityQueue with default initial capacity of 11 and all objects will be inserted as per default natural sorting order.
- *PriorityQueue p = new PriorityQueue(int initialCapacity);*
Creates an Empty PriorityQueue with specified initial capacity and all objects will be inserted as per default natural sorting order.
- *PriorityQueue p = new PriorityQueue(int initialCapacity, Comparator C);*
Creates an Empty PriorityQueue with specified initial capacity and all objects will be inserted as per customized sorting order.
- *PriorityQueue p = new PriorityQueue(SortedSet S);*
Creates an equivalent PriorityQueue of specified SortedSet S.
- *PriorityQueue p = new PriorityQueue(Collection C);*
Creates an equivalent PriorityQueue of specified Collection C.

Code Example

The image displays two screenshots of an IDE, likely IntelliJ IDEA, showing a Java program that demonstrates the use of a PriorityQueue. The code is in a file named Main.java.

Top Screenshot: The code defines a class PriorityQueueDemo with a main method. It creates a PriorityQueue p, prints its peek value (null), and then iterates from 0 to 10, offering each number to the queue. After each offer, it prints the queue's state, the next element to be polled, and the polled element itself. The output shows the queue elements in sorted order: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

```
1 import java.util.*;
2 class PriorityQueueDemo
3 {
4     public static void main(String[] args)
5     {
6         PriorityQueue p = new PriorityQueue();
7         System.out.println(p.peek());
8
9         for(int i=0; i<=10; i++)
10        {
11            p.offer(i);
12        }
13
14        System.out.println(p);
15        System.out.println(p.poll());
16        System.out.println(p);
17    }
18 }
```

Bottom Screenshot: The code is similar to the top one, but it includes a comment indicating that the poll operation throws an exception. The output shows the same sorted sequence of numbers, but with an exception message: "Exception in thread 'main' java.util.NoSuchElementException: java.base/java.util.AbstractQueue.element (AbstractQueue.java:136) at PriorityQueueDemo.main(PriorityQueueDemo.java:8)".

```
1 import java.util.*;
2 class PriorityQueueDemo
3 {
4     public static void main(String[] args)
5     {
6         PriorityQueue p = new PriorityQueue();
7         System.out.println(p.peek());
8         System.out.println(p.poll()); // Throws Exception
9         for(int i=0; i<=10; i++)
10        {
11            p.offer(i);
12        }
13
14        System.out.println(p);
15        System.out.println(p.poll());
16        System.out.println(p);
17    }
18 }
```

PriorityQueue With Custom Sorting Order.

Main.java	Run	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3 public int compare(Object o1, Object o2) 4 { 5 String s1 = o1.toString(); 6 String s2 = o2.toString(); 7 return s2.compareTo(s1); 8 } 9 } 10 class PriorityQueueDemo 11 { 12 public static void main(String[] args) 13 { 14 PriorityQueue p = new PriorityQueue(15, new MyComparator()); 15 p.offer("A"); 16 p.offer("Z"); 17 p.offer("L"); 18 p.offer("B"); 19 p.offer("X"); 20 System.out.println(p); 21 } 22 }</pre>		<pre>java -cp /tmp/qHscE9rI9F PriorityQueueDemo [Z, X, L, A, B]</pre>

Note: Some platforms don't provide support for Thread Priorities and Priority Queues.



7.. Map

- Map is not the Child Interface of Collection.
- If we want to represent a group of objects as a Key-Value Pairs then we should go for Map Interface.
- In Map, both Key and Value are Objects only where we cannot have duplicate Keys but we can have duplicate Values.
- There are some classes which implements Map Interface:
 1. HashMap
 2. LinkedHashMap
 3. WeakHashMap
 4. IdentityHashMap
 5. Hashtable is also a Child Class of Dictionary which is an abstract Class
 6. Properties
- Each Key-Value Pair in Map is called Entry. Hence Map is considered as a collection of Entry Objects.

Methods Defined by Map Interface

- **Object put(Object key, Object Value)**
Add a key-value pair i.e., entry in a Map. If the key is already present then old value will be replaced with new value and returns old value. If there is no replacement happening then it will returns null.
- **void putAll(Map m)**
Add all the entries present in a Map M to the Map object on which it is called.
- **Object get(Object key)**
Returns the Value associated with specified key. If specified key is not present in a Map then it will return null.
- **Object remove(Object key)**
Remove an entry present in a map with specified key.
- **boolean containsKey(Object key)**
Returns true if specified key is present in a map else return false.
- **boolean containsValue(Object Value)**
Returns true if specified value is present in a Map else returns false.
- **boolean isEmpty()**
Returns true if Map is empty else returns false.
- **int size()**
Returns the number of entries present in a map.
- **void clear()**
Remove all the entries from map.
- **Set keySet()**
Returns the set of Keys present in Map.

- **Collection values()**
Returns the collection of Values present in Map.
- **Set entrySet()**
Returns the Set of all entries present in Map.

Entry Interface

- A map is a group of key-value pairs and each key-value pair is called an entry. Hence, Map is considered as a collection of Entry Objects.
- Without existing Map object there is no change of existing of Entry Object. Hence, Entry Interface is defined inside Map Interface.

Methods defined in Entry Interface

These methods can be applied only on Entry Objects

- **Object getKey()**
Returns the Key for an Entry
- **Object getValue()**
Returns the Value of an Entry
- **Object setValue(Object newValue)**
Replace the Value of an Entry with newValue

7.1 HashMap

- The Underlying data structure for HashMap is Hashtable.
- Insertion order is not preserved and insertion is based on Hash code of Keys.
- Duplicate Keys are not allowed but duplicate values are allowed.
- Both Keys and Values can be both homogenous and heterogeneous.
- Key can be null and can be inserted only once while null values can be added any number of times.
- Implements Serializable, Cloneable but doesn't implement RandomAccess Interface.
- HashMap is best choice when our frequent operation is search operation.

Constructors of HashMap

- *HashMap m = new HashMap();*
Creates an empty HashMap object with default initial capacity 16 and default fill ratio of 0.75.
- *HashMap m = new HashMap(int initialCapacity);*
Creates an empty HashMap Object with specified initial capacity but default fill ratio of 0.75.
- *HashMap m = new HashMap(int initialCapacity, float fillRatio);*
Creates an empty HashMap Object with specified initial capacity and specified fill ratio.
- *HashMap m = new HashMap(Map M);*
Creates an equivalent HashMap of Specified Map M.

Code Example

```
Main.java
1- import java.util.*;
2- class HashMapDemo{
3-     public static void main(String[] args){
4         HashMap m = new HashMap();
5         System.out.println(m.isEmpty());
6         System.out.println(m.size());
7         m.put("101","Vikash");
8         m.put("102","Naina");
9         m.put("103",28);
10        m.put("104",45.54);
11        System.out.println(m);
12        m.put("102","Tripti");
13        m.put(null, null);
14        m.put("105", null);
15        System.out.println(m);
16        System.out.println(m.isEmpty());
17        System.out.println(m.size());
18        System.out.println(m.get("101"));
19        System.out.println(m.size());
20        System.out.println(m.containsKey("101"));
21        System.out.println(m.containsValue("Vikash"));
22        m.remove(null);
23        System.out.println(m);
24        System.out.println(m.keySet());
25        System.out.println(m.values());
26        Set entry = m.entrySet();
27        System.out.println(entry);
28        Iterator itr = entry.iterator();
29        while(itr.hasNext()){
30            Map.Entry ma =(Map.Entry) itr.next();
31            System.out.println(ma.getKey() + "----" + ma.getValue());
32        }
33        System.out.println(m.get("108"));
34    }
35 }
```

```
Output
java -cp /tmp/108875vF2w HashMapDemo
true
0{101=Vikash, 102=Naina, 103=28, 104=45.54}
{null=null, 101=Vikash, 102=Tripti, 103=28, 104=45.54, 105=null}
false
6
Vikash
6
true
true
{101=Vikash, 102=Tripti, 103=28, 104=45.54, 105=null}
[101, 102, 103, 104, 105]
[Vikash, Tripti, 28, 45.54, null]
[101=Vikash, 102=Tripti, 103=28, 104=45.54, 105=null]
101----Vikash
102----Tripti
103----28
104----45.54
105----null
null
```

Difference between HashMap and Hashtable

- In HashMap None of the Method is Synchronized while In Hashtable Every Method is Synchronized
- Multiple Threads are allowed to operate at the same time on the HashMap Object. Hence it is Not Thread-safe while Only one Thread is allowed to operate at the same time on the Hashtable Object. Hence, it is Thread-Safe.
- Since there is no waiting time for the threads so performance is high for HashMap while Since there is waiting time for the threads so performance is low for Hashtable.
- Null key or Null Values are allowed in HashMap while Not Allowed in Hashtable
- HashMap is not legacy introduced in 1.2 Version of Java while Hashtable is legacy and introduced in 1.0 Version of Java

7.2 LinkedHashMap

- LinkedHashMap is a child class of HashMap.
- The underlying data structure for LinkedHashMap is Linked List and Hashtable.
- Since it uses Linked List and Hash Table both, Insertion Order is Preserved.
- Except this above functionality, it is similar to HashMap including constructors and methods.
- LinkedHashMap is mainly used to create Cache Based Applications.

Code Example

Main.java	<pre>1- import java.util.*; 2 class LinkedHashMapDemo 3 { 4- public static void main(String[] args) { 5 HashMap h = new HashMap(); 6 h.put("Salman",101); 7 h.put("Arjun",102); 8 h.put("Guru",103); 9 h.put("Bacchan",104); 10 System.out.println(h); 11 12 LinkedHashMap l = new LinkedHashMap(); 13 l.put("Salman",101); 14 l.put("Arjun",102); 15 l.put("Guru",103); 16 l.put("Bacchan",104); 17 System.out.println(l); 18 } 19 }</pre>	Output <pre>java -cp /tmp/GiNfdQQZYB LinkedHashMapDemo {Guru=103, Bacchan=104, Salman=101, Arjun=102} {Salman=101, Arjun=102, Guru=103, Bacchan=104}</pre>
-----------	---	---

7.3 IdentityHashMap

- IdentityHashMap is a child of HashMap class.
- In IdentityHashMap, JVM will use == Operator to identify duplicates which is meant for address or reference comparison, while in HashMap, JVM will use equals() method to identify duplicates which is meant for content comparison. Apart from these, it is exactly same as HashMap including methods and Constructors.

Code Example

Main.java	<pre>1- import java.util.*; 2 class LinkedHashMapDemo 3 { 4- public static void main(String[] args) { 5 Integer I1 = new Integer(10); 6 Integer I2 = new Integer(10); 7 HashMap h = new HashMap(); 8 h.put(I1,"Vikash"); 9 h.put(I2,"Sharma"); 10 System.out.println(h); 11 12 IdentityHashMap i = new IdentityHashMap(); 13 i.put(I1,"Vikash"); 14 i.put(I2,"Sharma"); 15 System.out.println(i); 16 } 17 }</pre>	Output <pre>java -cp /tmp/GiNfdQQZYB LinkedHashMapDemo {10=Sharma} {10=Vikash, 10=Sharma}</pre>
-----------	---	--

7.4 WeakHashMap

- WeakHashMap is a child class of HashMap class.
- It is exactly same as HashMap except the following difference. In HashMap, though the Object doesn't have any reference, it is not eligible for Garbage Collector if it is associated with HashMap. i.e., HashMap dominates Garbage Collector. But in case of WeakHashMap, if Object doesn't contain any reference, is eligible for Garbage Collector even though associated with WeakHashMap i.e., Garbage Collector Dominates Weak HashMap.

Code Example

```
Main.java
1- import java.util.*;
2- class Test
3- {
4-     public String toString()
5-     {
6-         return "temp";
7-     }
8-     public void finalize()
9-     {
10-         System.out.println("Garbage Collector Called");
11-     }
12- }
13- class WeakHashMapDemo
14- {
15-     public static void main(String[] args) throws InterruptedException {
16-         HashMap h = new HashMap();
17-         Test t1 = new Test();
18-         h.put(t1, "Vikash");
19-         System.out.println(h);
20-         t1=null;
21-         System.gc();
22-         Thread.sleep(5000);
23-         System.out.println(h);
24-
25-         Test t2 = new Test();
26-         WeakHashMap w = new WeakHashMap();
27-         w.put(t2, "Vikash");
28-         System.out.println(w);
29-         t2=null;
30-         System.gc();
31-         Thread.sleep(5000);
32-         System.out.println(w);
33-     }
34- }
```

```
Output
java -cp /tmp/GiNfdQQZYB WeakHashMapDemo
{temp=Vikash}
{temp=Vikash}
{temp=Vikash}
Garbage Collector Called
{}

```

7.5 SortedMap

- SortedMap is a Child of Map Interface.
- If we want to represent a group of objects as a Key-Value Pairs where insertion of Object is done according to some sorting order of Keys then we should go for SortedMap. In SortedMap sorting should be based on Keys not based on Values.

Method Specific to SortedMap

- **Object firstKey()**
Returns the First Key of SortedMap
- **Object lastKey()**
Returns the Last Key of SortedMap
- **SortedMap headMap(Object key)**
Return the SortedMap whose keys are < Specified Key
- **SortedMap tailMap(Object key)**
Return the SortedMap whose keys are >= Specified Key
- **SortedMap subMap(Object K1, Object K2)**
Return the SortedMap whose keys are >= K1 Key and < K2 Key.
- **Comparator comparator()**
Returns Comparator Object that describes underlying sorting technique. If we are using default natural sorting order then we will get null. For numbers default sorting order is ascending order while for string it is Alphabetical Order.

7.6 TreeMap

- The Underlying data structure for TreeMap is Red-Black Tree.
- Insertion order is not preserved and all insertions will be done on the sorting order of Keys not on values.
- Duplicates Keys are not allowed but duplicate values are allowed.
- Heterogenous Objects are not allowed if we are using a default natural sorting order, it will throw **ClassCastException** but if we are defining a customized sorting order then Heterogenous Objects are allowed.

Null Acceptance in TreeMap

- For Non-empty TreeMap if we are trying to add null key then we are going to get **NullPointerException**.
- For Empty TreeMap as a first key element, null is allowed to insert until Java version 1.6. But after inserting null if we are trying to insert any other Object then we will get **NullPointerException**. But after java version 1.7 Null is not allowed to enter into TreeMap as a first key element also.
- For null values there is no restriction.

Main.java	Output
<pre>1- import java.util.*; 2- class TreeMapNullKey 3- { 4- public static void main(String[] args) 5- { 6- TreeMap m = new TreeMap(); 7- m.put("A",null); 8- System.out.println(m); 9- m.put(null,"A"); 10 System.out.println(m); 11 } 12 }</pre>	<pre>java -cp /tmp/jp6pK8kvY0 TreeMapNullKey {A=null} Exception in thread "main" java.lang.NullPointerException at java.base/java.util.TreeMap.put(TreeMap.java:561) at TreeMapNullKey.main(TreeMapNullKey.java:9)</pre>

Constructors of TreeMap

- *TreeMap m = new TreeMap();*
Creates an empty TreeMap m with default natural sorting order of Keys.
- *TreeMap m = new TreeMap(Comparator C);*
Creates an empty TreeMap m with Customized Sorting Order. The Customized Sorting order can be implemented by using Comparator Object C.
- *TreeMap m = new TreeMap(Map M);*
Creates an equivalent TreeMap of specified Map M with default natural sorting Order.
- *TreeMap m = new TreeMap(SortedMap M);*
Creates a TreeMap equivalent to SortedMap M with Same sorting order which M carries.

Code Example

Default Natural Sorting Order

Main.java	Output
<pre>1- import java.util.*; 2 class TreeMapNullKey 3 { 4 public static void main(String[] args) 5 { 6 TreeMap m = new TreeMap(); 7 m.put(101,"Vikash"); 8 m.put(104,"Naina"); 9 m.put(100,"Akash"); 10 m.put(106,"Pooja"); 11 m.put(103,"Payal"); 12 System.out.println(m); 13 System.out.println(m.firstKey()); 14 System.out.println(m.lastKey()); 15 System.out.println(m.headMap(104)); 16 System.out.println(m.tailMap(104)); 17 System.out.println(m.subMap(102,106)); 18 } 19 }</pre>	<pre>java -cp /tmp/JpGpK8kVy0 TreeMapNullKey {100=Akash, 101=Vikash, 103=Payal, 104=Naina, 106=Pooja} 100 106 {100=Akash, 101=Vikash, 103=Payal} {104=Naina, 106=Pooja} {103=Payal, 104=Naina}</pre>

With Comparator default natural sorting order i.e., ascending order.

Main.java	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3 public int compare(Object o1, Object o2) 4 { 5 String s1 = o1.toString(); 6 String s2 = o2.toString(); 7 return s1.compareTo(s2); 8 } 9 } 10 class TreeMapSorting 11 { 12 public static void main(String[] args) 13 { 14 TreeMap m = new TreeMap(new MyComparator()); 15 m.put("XXX",10); 16 m.put("AAA",20); 17 m.put("ZZZ",30); 18 m.put("LLL",40); 19 System.out.println(m); 20 } 21 }</pre>	<pre>java -cp /tmp/PbXA3DJcVZ TreeMapSorting {AAA=20, LLL=40, XXX=10, ZZZ=30}</pre>

With Comparator customized sorting order i.e., descending order.

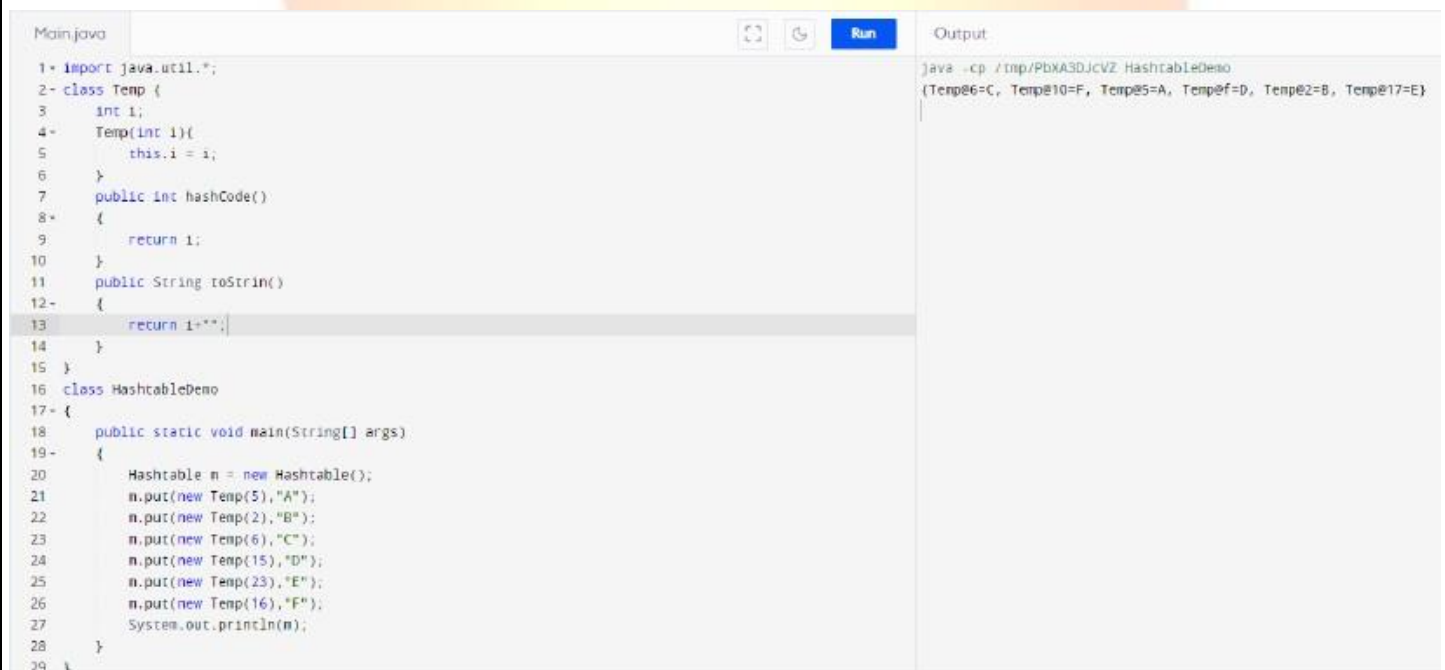
Main.java	Output
<pre>1- import java.util.*; 2- class MyComparator implements Comparator{ 3 public int compare(Object o1, Object o2) 4 { 5 String s1 = o1.toString(); 6 String s2 = o2.toString(); 7 return s2.compareTo(s1); 8 } 9 } 10 class TreeMapSorting 11 { 12 public static void main(String[] args) 13 { 14 TreeMap m = new TreeMap(new MyComparator()); 15 m.put("XXX",10); 16 m.put("AAA",20); 17 m.put("ZZZ",30); 18 m.put("LLL",40); 19 System.out.println(m); 20 } 21 }</pre>	<pre>java -cp /tmp/PbXA3DJcVZ TreeMapSorting {ZZZ=30, XXX=10, LLL=40, AAA=20}</pre>

7.7 Hashtable

- The underlying data structure for Hashtable is Hash Table.
- Insertion order is not preserved and insertion is based on Hash code of Keys.
- Duplicate keys are not allowed but values can be duplicated.
- Heterogenous Objects are allowed for both keys and values.
- Neither null key nor null value insertion is allowed.
- Implements Serializable, Clonable but not RandomAccess Interface.
- Every method present in Hashtable is synchronized and hence Hashtable is considered to be the Thread Safe.
- Hashtable is a best choice if our frequent operation is a search operation.

Constructors of Hashtable

- *Hashtable h= new Hashtable();*
Creates an empty Hashtable object with default initial capacity 11 and default fill ratio of 0.75.
- *Hashtable h= new Hashtable(int initialCapacity);*
Creates an empty Hashtable Object with specified initial capacity but default fill ratio of 0.75.
- *Hashtable h= new Hashtable(int initialCapacity, float fillRatio);*
Creates an empty Hashtable Object with specified initial capacity and specified fill ratio.
- *Hashtable h= new Hashtable(Map M);*
Creates an equivalent Hashtable of Specified Map M.



The screenshot shows an IDE with a file named 'Main.java'. The code defines a 'Temp' class with an 'int i' attribute and methods 'hashCode()' and 'toString()'. The 'hashCode()' method returns the value of 'i'. The 'toString()' method returns 'i+'. A 'HashtableDemo' class contains a 'main' method that creates a 'Hashtable' and inserts six entries: (5, 'A'), (2, 'B'), (6, 'C'), (15, 'D'), (23, 'E'), and (16, 'F'). The output window shows the result of running the program: 'java -cp /tmp/PbXA3DJCVZ HashtableDemo (Temp@6=C, Temp@10=F, Temp@5=A, Temp@f=D, Temp@2=B, Temp@17=E)'. The output is unordered, demonstrating that the insertion order is not preserved in a Hashtable.

```
1- import java.util.*;
2- class Temp {
3-     int i;
4-     Temp(int i){
5-         this.i = i;
6-     }
7-     public int hashCode()
8-     {
9-         return i;
10-    }
11-    public String toString()
12-    {
13-        return i+" ";
14-    }
15- }
16- class HashtableDemo
17- {
18-     public static void main(String[] args)
19-     {
20-         Hashtable m = new Hashtable();
21-         m.put(new Temp(5), "A");
22-         m.put(new Temp(2), "B");
23-         m.put(new Temp(6), "C");
24-         m.put(new Temp(15), "D");
25-         m.put(new Temp(23), "E");
26-         m.put(new Temp(16), "F");
27-         System.out.println(m);
28-     }
29- }
```

Output

```
java -cp /tmp/PbXA3DJCVZ HashtableDemo
(Temp@6=C, Temp@10=F, Temp@5=A, Temp@f=D, Temp@2=B, Temp@17=E)
```

If we change the Code in hashCode() method then output will also change.

Main.java	Output
<pre>1- import java.util.*; 2- class Temp { 3- int i; 4- Temp(int i){ 5- this.i = i; 6- } 7- public int hashCode() 8- { 9- return i%9; 10- } 11- public String toString() 12- { 13- return i+""; 14- } 15- } 16- class HashtableDemo 17- { 18- public static void main(String[] args) 19- { 20- Hashtable n = new Hashtable(); 21- n.put(new Temp(5),"A"); 22- n.put(new Temp(2),"B"); 23- n.put(new Temp(6),"C"); 24- n.put(new Temp(15),"D"); 25- n.put(new Temp(23),"E"); 26- n.put(new Temp(16),"F"); 27- System.out.println(n); 28- } 29- }</pre>	<pre>java -cp /tmp/PbXA3DjCVZ HashtableDemo {Temp@7=F, Temp@6=D, Temp@6=C, Temp@5=E, Temp@5=A, Temp@2=B}</pre>

Now Question Arises How Hashtable Works internally?

- Whenever we call `Hashtable t = new Hashtable();` it will create an empty Hashtable with default initial capacity of 11. i.e., it will create 11 divisions called buckets in Hashtable starting from 0 to `initialCapacity-1`.
- Whenever we call `t.put("Key","Value");` method, internally JVM will calculate `hashCode` of Key provided. It can use either predefined `hashCode()` method or our Customized `HashCode` value. Whatever the value of `hashCode` we get, that particular value will be stored at that bucket of Hashtable.
- If `hashCode` value of Key is greater than Hashtable capacity then `hashCode = hashCode % Capacity of Hashtable`.
- Whenever we get same `hashCode` for multiple keys then that entry will be made on the same bucket.
- Once every entry is done, while printing, it will print values from top to down buckets and if multiple entries in same buckets then right to left will be printed.

Let's change Hashtable capacity and then see the output.

Main.java	Output
<pre>1- import java.util.*; 2- class Temp { 3- int i; 4- Temp(int i){ 5- this.i = i; 6- } 7- public int hashCode() 8- { 9- return i%9; 10- } 11- public String toString() 12- { 13- return i+""; 14- } 15- } 16- class HashtableDemo 17- { 18- public static void main(String[] args) 19- { 20- Hashtable n = new Hashtable(25); 21- n.put(new Temp(5),"A"); 22- n.put(new Temp(2),"B"); 23- n.put(new Temp(6),"C"); 24- n.put(new Temp(15),"D"); 25- n.put(new Temp(23),"E"); 26- n.put(new Temp(16),"F"); 27- System.out.println(n); 28- } 29- }</pre>	<pre>java -cp /tmp/PbXA3DjCVZ HashtableDemo {Temp@7=F, Temp@6=D, Temp@6=C, Temp@5=E, Temp@5=A, Temp@2=B}</pre>

7.8 Properties

- In a Java program, if any value or anything changes very frequently then that value shouldn't be hardcoded in the source code of a program. Because if we do so then we have to recompile, rebuild, redeploy and sometime restart the server which is quite time consuming and creates a bad impact on the client. That value should be placed in a properties files. Like Database Configuration User name and URL etc.
- We can pass these properties from properties file into Java Program very easily by using Properties class object which hold properties coming from properties file.
- In Java Properties is similar to other Map like HashMap which can store key-value pairs. But in Properties, both Key and Value should be of String type.
- The advantage of this approach is if we want to make any change to any of the properties then we can do and just redeploy our code.

Constructor of Properties

- *Properties p = new Properties();*

Methods Specific to Properties

- **String getProperty(String name)**
Get Value of Specified Property Name
- **String setProperty(String name, String value)**
Add a new Property and replace the Property Value if Property Name already exists.
- **Enumeration propertyNames()**
Provides all property names.
- **void load(InputStream is)**
It loads the Properties defined in Properties file into the Properties Class Object.
- **void store(OutputStream os, String comment)**
After adding Properties by setProperty() method, this method will store the properties from properties object into properties file.

```
Main.java  [Icons]  Run
1 import java.util.*;
2 import java.io.*;
3 class PropertiesDemo
4 {
5     public static void main(String[] args) throws Exception
6     {
7         Properties p = new Properties();
8         FileInputStream fis = new FileInputStream("abc.properties");
9         p.load(fis);
10        System.out.println(p);
11        System.out.println(p.getProperty("venky"));
12        p.setProperty("uName", "Vikash");
13        FileOutputStream fos = new FileOutputStream("abc.properties");
14        p.store(fos, "Updated By Vikash Sharma");
15    }
16 }
```


7.9 NavigableMap

- From Java 1.6 Version NavigableMap Interface was Introduced
- NavigableMap is a Child of SortedMap Interface. It contains several methods for Navigation Purposes.
- There is only one implementation class for NavigableMap which TreeMap.

Methods Introduced by NavigableMap

- **Object floorKey(e)**
Returns highest key element which is $\leq e$
- **Object lowerKey(e)**
Returns highest key element which is $< e$
- **Object ceilingKey(e)**
Returns Lowest key element which is $\geq e$
- **Object higherKey(e)**
Returns Lowest key element which is $> e$
- **pollFirstEntry()**
Remove and Return first entry.
- **pollLastEntry()**
Remove and Return last entry.
- **NavigableMap descendingMap()**
Return NavigableMap in reverse Order.

Code Example

Main.java	Run	Output
<pre>1- import java.util.*; 2 class NavigableMapDemo 3 { 4 public static void main(String[] args) 5 { 6 TreeMap<Integer,String> t = new TreeMap<>(); 7 t.put(1000,"Vikash"); 8 t.put(2000,"Naina"); 9 t.put(3000,"Naitik"); 10 t.put(4000,"Navika"); 11 t.put(5000,"Akash"); 12 System.out.println(t); 13 System.out.println(t.ceilingKey(2000)); 14 System.out.println(t.higherKey(2000)); 15 System.out.println(t.floorKey(3000)); 16 System.out.println(t.lowerKey(2000)); 17 System.out.println(t.pollFirstEntry()); 18 System.out.println(t.pollLastEntry()); 19 System.out.println(t.descendingMap()); 20 System.out.println(t); 21 } 22 }</pre>		<pre>java -cp /tmp/506QY1xbbg NavigableMapDemo {1000=Vikash, 2000=Naina, 3000=Naitik, 4000=Navika, 5000=Akash} 2000 3000 3000 1000 1000=Vikash 5000=Akash {4000=Navika, 3000=Naitik, 2000=Naina} {2000=Naina, 3000=Naitik, 4000=Navika}</pre>

8.. Collections

- Collections class defines several utility methods for Collection Objects like Sorting, searching, reversing etc.

Sorting Methods Provided by Collections Utility Class

- **public static void sort(List l)**
Sort the elements of list as per default natural sorting order. Since we are depending on default natural sorting order, so all the elements of a List should be Homogenous and Comparable else we will get ClassCastException. Also if try to add null then we will get NullPointerException.
- **public static void sort(List l, Comparator C)**
Sort the elements of list as per customized sorting order by using Comparator Object.

Code Example Of sort() method



The first screenshot shows a Java file named Main.java with the following code:

```
1- import java.util.*;
2 class CollectionSortDemo
3- {
4     public static void main(String[] args)
5     {
6         ArrayList<String> l = new ArrayList<String>();
7         l.add("Z");
8         l.add("A");
9         l.add("K");
10        l.add("N");
11        System.out.println(l);
12        Collections.sort(l);
13        System.out.println(l);
14    }
15 }
```

The output of this code is:

```
java -cp /tmp/765mn0uqd3 CollectionSortDemo
[Z, A, K, N]
[A, K, N, Z]
```

The second screenshot shows a Java file named Main.java with the following code:

```
1- import java.util.*;
2 class MyComparator implements Comparator
3- {
4     public int compare(Object O1, Object O2)
5     {
6         String S1 = O1.toString();
7         String S2 = O2.toString();
8         return S2.compareTo(S1);
9     }
10 }
11 class CollectionSortDemo
12- {
13     public static void main(String[] args)
14     {
15         ArrayList<String> l = new ArrayList<String>();
16         l.add("Z");
17         l.add("A");
18         l.add("K");
19         l.add("N");
20         System.out.println(l);
21         Collections.sort(l, new MyComparator());
22         System.out.println(l);
23     }
24 }
```

The output of this code is:

```
java -cp /tmp/765mn0uqd3 CollectionSortDemo
[Z, A, K, N]
[Z, N, K, A]
```



Searching method provided by Collections class

- **public static int binarySearch(List L, Object target);**
Search a target Object in a Sorted List. Returns an index value for positive search results while returns an Insertion point i.e., a index at which we can put target Object in a Sorted List. This method internally uses binary search algorithm for which it is compulsory to have

an input list as sorted. This method is used if list is sorted as per default natural sorting order.

- **public static int binarySearch(List L, Object target, Comparator C);**
This method is used if List L is sorted as per Customized Sorting Order.

Code Example of BinarySearch

Main.java	Run	Output
<pre>1+ import java.util.*; 2 class SearchDemo 3+ { 4 public static void main(String[] args) 5 { 6 ArrayList L = new ArrayList(); 7 L.add("Z"); 8 L.add("A"); 9 L.add("M"); 10 L.add("K"); 11 L.add("a"); 12 System.out.println(L); 13 Collections.sort(L); 14 System.out.println(L); 15 System.out.println(Collections.binarySearch(L,"M")); 16 System.out.println(Collections.binarySearch(L,"B")); 17 } 18 }</pre>		<pre>java -cp /tmp/vL1bA2pajm SearchDemo [Z, A, M, K, a] [A, K, M, Z, a] 2 -2</pre>
<pre>1+ import java.util.*; 2 class MyComparator implements Comparator 3+ { 4 public int compare(Object O1, Object O2) 5 { 6 String s1 = O1.toString(); 7 String s2 = O2.toString(); 8 return s2.compareTo(s1); 9 } 10 } 11 class SearchDemo 12+ { 13 public static void main(String[] args) 14 { 15 ArrayList L = new ArrayList(); 16 L.add("Z"); 17 L.add("A"); 18 L.add("M"); 19 L.add("K"); 20 L.add("a"); 21 System.out.println(L); 22 MyComparator c = new MyComparator(); 23 Collections.sort(L, c); 24 System.out.println(L); 25 System.out.println(Collections.binarySearch(L,"M",c)); 26 System.out.println(Collections.binarySearch(L,"B",c)); 27 } 28 }</pre>		<pre>java -cp /tmp/vL1bA2pajm SearchDemo [Z, A, M, K, a] [a, Z, M, K, A] 2 -5</pre>

For a List of N elements:

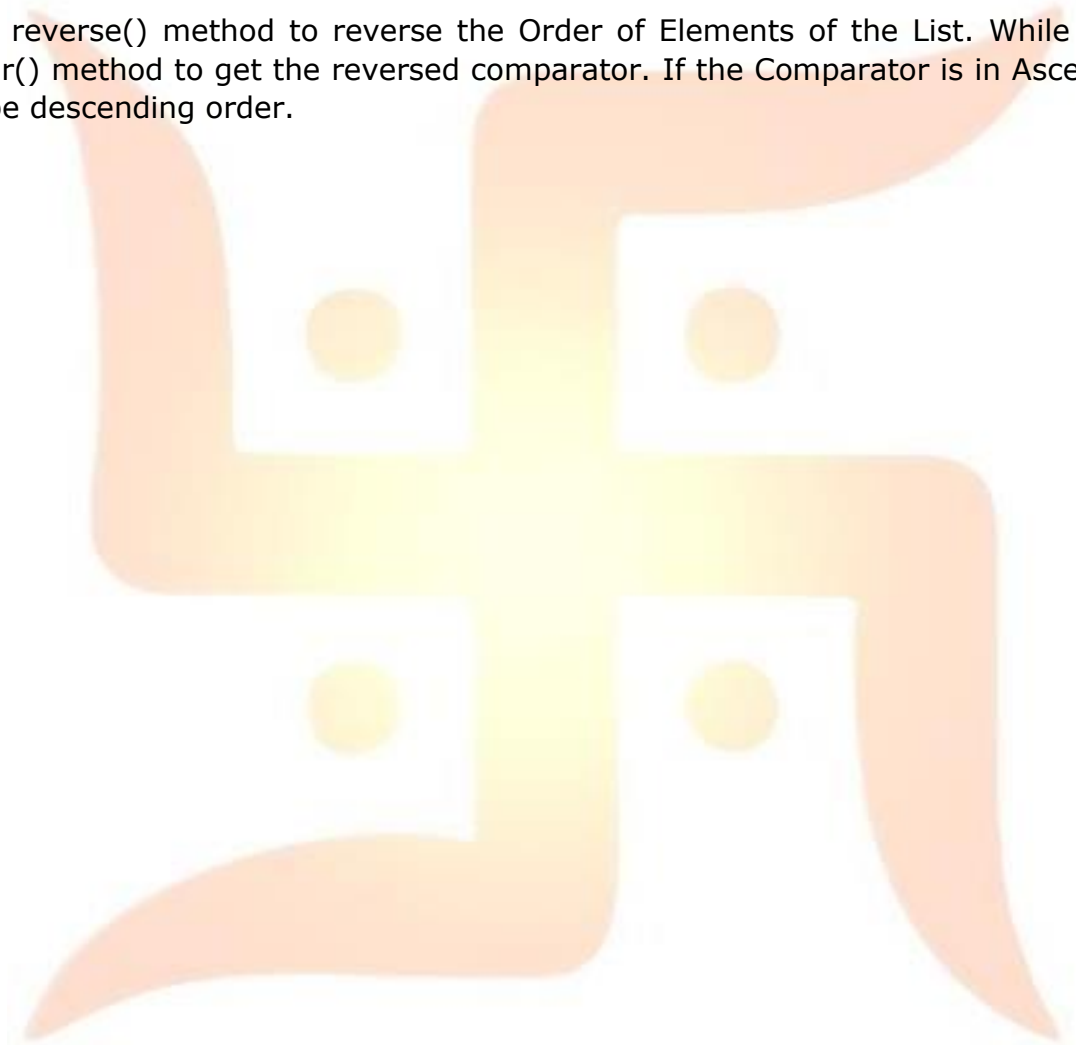
- Successful Search Result Range: 0 to N-1
- Unsuccessful Search Result Range: -(N+1) to -1
- Total Result Range: -(N+1) to N-1

Reversing Methods provided by Collections Class

- **public static void reverse(List L);**

Main.java	Run	Output
<pre>1- import java.util.*; 2 class ReverseDemo 3- { 4 public static void main(String[] args) 5- { 6 ArrayList L = new ArrayList(); 7 L.add("Z"); 8 L.add("A"); 9 L.add("M"); 10 L.add("K"); 11 L.add("a"); 12 System.out.println(L); 13 Collections.reverse(L); 14 System.out.println(L); 15 } 16 }</pre>		<pre>java -cp /tmp/vL1bA2pajm ReverseDemo [Z, A, M, K, a] [a, K, M, A, Z]</pre>

We can use reverse() method to reverse the Order of Elements of the List. While we can use reverseOrder() method to get the reversed comparator. If the Comparator is in Ascending Order output will be descending order.

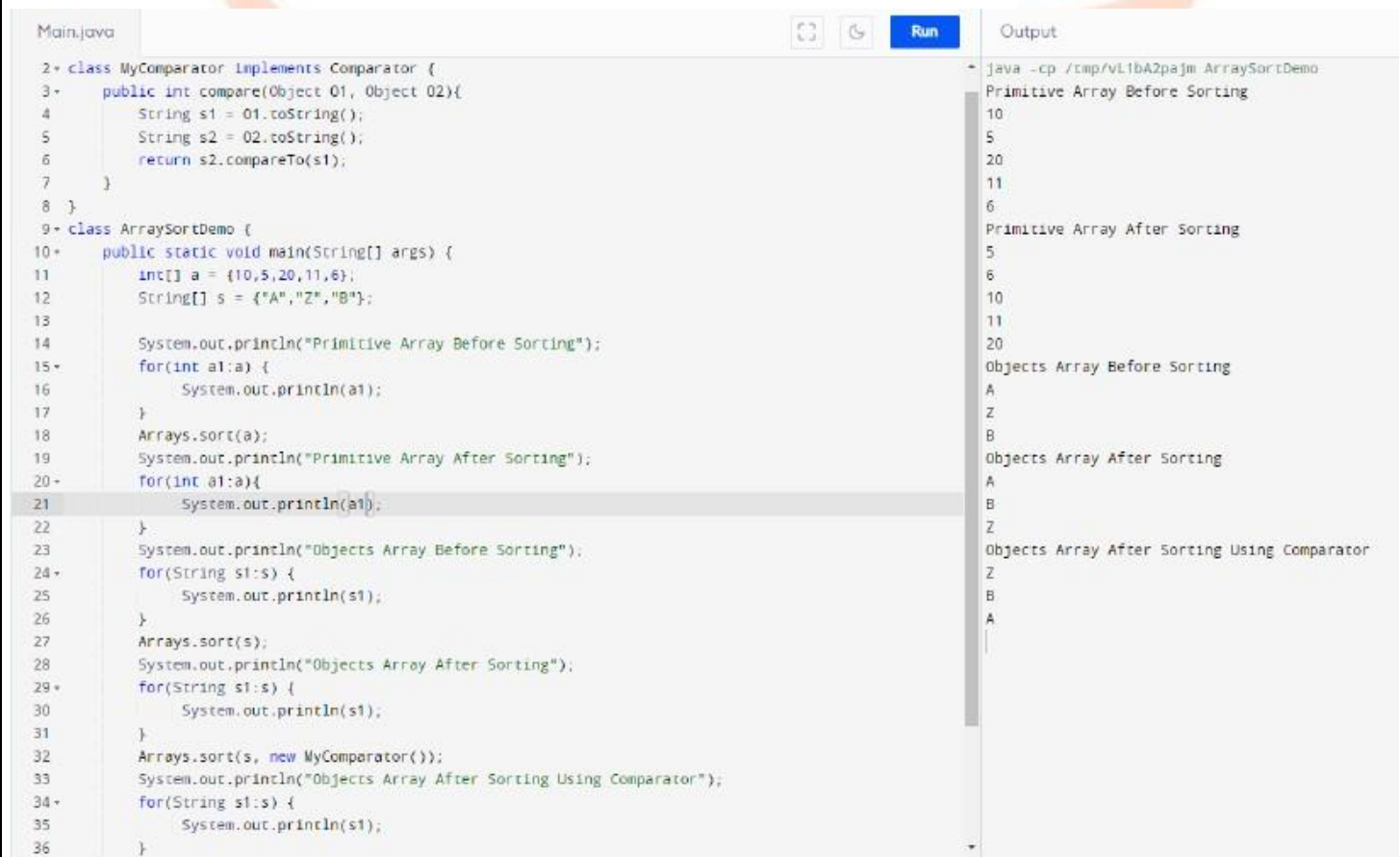


9.. Arrays

- It is a Utility Class to define several utility methods for Array Objects or Arrays.

Sorting Functions of Arrays

- public static void sort(primitive[] p)**
Sort Primitive Array According to Default Natural Sorting Order
- public static void sort(Object[] o)**
Sort Object Array According to Default Natural Sorting Order
- public static void sort(Object[] o, Comparator C)**
Sort Object Array According to Custom Order.



The screenshot shows an IDE with a file named 'Main.java'. The code defines a custom comparator 'MyComparator' and a class 'ArraySortDemo'. The 'main' method in 'ArraySortDemo' demonstrates sorting a primitive integer array and a string object array using the default 'Arrays.sort()' method, and then sorting the string array using the custom 'MyComparator'.

```
2- class MyComparator implements Comparator {
3-     public int compare(Object O1, Object O2){
4-         String s1 = O1.toString();
5-         String s2 = O2.toString();
6-         return s2.compareTo(s1);
7-     }
8- }
9- class ArraySortDemo {
10-     public static void main(String[] args) {
11-         int[] a = {10,5,20,11,6};
12-         String[] s = {"A","Z","B"};
13-
14-         System.out.println("Primitive Array Before Sorting");
15-         for(int a1:a) {
16-             System.out.println(a1);
17-         }
18-         Arrays.sort(a);
19-         System.out.println("Primitive Array After Sorting");
20-         for(int a1:a){
21-             System.out.println(a1);
22-         }
23-         System.out.println("Objects Array Before Sorting");
24-         for(String s1:s) {
25-             System.out.println(s1);
26-         }
27-         Arrays.sort(s);
28-         System.out.println("Objects Array After Sorting");
29-         for(String s1:s) {
30-             System.out.println(s1);
31-         }
32-         Arrays.sort(s, new MyComparator());
33-         System.out.println("Objects Array After Sorting Using Comparator");
34-         for(String s1:s) {
35-             System.out.println(s1);
36-         }
37-     }
38- }
```

The output window shows the following results:

```
java -cp ./tmp/vL1bA2paJm ArraySortDemo
Primitive Array Before Sorting
10
5
20
11
6
Primitive Array After Sorting
5
6
10
11
20
Objects Array Before Sorting
A
Z
B
Objects Array After Sorting
A
B
Z
Objects Array After Sorting Using Comparator
Z
B
A
```

Note: We can sort primitive arrays only according to Default natural sorting order while Object arrays can be sorted both according to default and customized sorting order.

Searching Elements in an Array

- public static int binarySearch(Primitive[] P, Primitive target);**
Search a primitive target in a Sorted Primitive Array. Returns an index value for positive search results while returns an Insertion point i.e., a index at which we can put primitive target in a Sorted Array. This method internally uses binary search algorithm for which it is compulsory to have an input array as sorted. This method is used if array is sorted as per default natural sorting order.

- `public static int binarySearch(Object[] O, Object target);`
Search a target Object in a Sorted Object Array. Returns an index value for positive search results while returns an Insertion point i.e., a index at which we can put target object in a Sorted object Array. This method internally uses binary search algorithm for which it is compulsory to have an input array as sorted. This method is used if array is sorted as per default natural sorting order.
- `public static int binarySearch(Object[] O, Object target, Comparator C);`
This method is used if Object Array O is sorted as per Customized Sorting Order.

Note: All rules of Arrays class `binarySearch()` method is same as Collections class `binarySearch()` method.

Main.java	Output
<pre> 1- import java.util.*; 2- class MyComparator implements Comparator { 3- public int compare(Object O1, Object O2){ 4- String s1 = O1.toString(); 5- String s2 = O2.toString(); 6- return s2.compareTo(s1); 7- } 8- } 9- class ArraySortDemo { 10- public static void main(String[] args) { 11 int[] a = {10,5,20,11,6}; 12 String[] s = {"A","Z","B"}; 13 Arrays.sort(a); 14 System.out.println(Arrays.binarySearch(a,20)); 15 System.out.println(Arrays.binarySearch(a,12)); 16 Arrays.sort(s); 17 System.out.println(Arrays.binarySearch(s,"B")); 18 System.out.println(Arrays.binarySearch(s,"D")); 19 MyComparator c = new MyComparator(); 20 Arrays.sort(s,c); 21 System.out.println(Arrays.binarySearch(s,"B",c)); 22 System.out.println(Arrays.binarySearch(s,"D",c)); 23 } 24 } </pre>	<pre> java -cp /tmp/vL1bA2paJm ArraySortDemo 4 -5 1 -3 1 -2 </pre>

Conversion of Array to List

- `public static List asList(Object[] O);`

Strictly speaking, this method won't create any independent List Object. For existing array we are getting List View.

By using array reference, if we perform any changes then that change will get reflected in List too similarly if we make any changes by using List reference same will get reflected in Array as well.

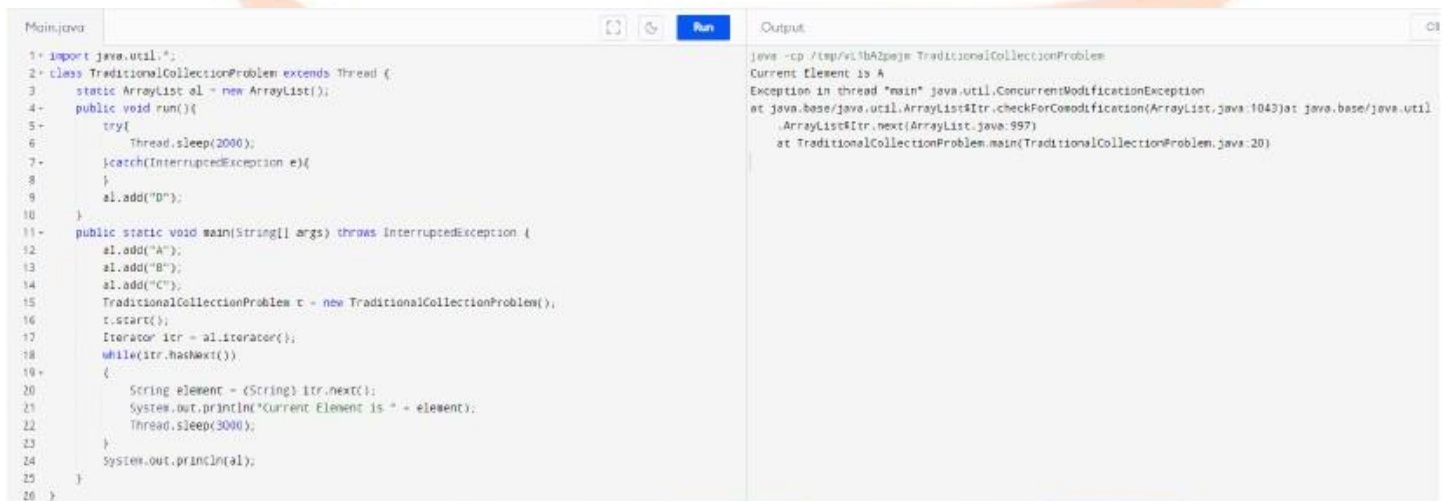
By using List reference, we can't perform any operation which increases or decreases the Size. If we try to do so, we will get **UnsupportedOperationException**.

By using list reference, if we try to update any element and trying to insert Heterogenous Object then we will get Run Time Exception Saying **ArrayStoreException**.

10.. Concurrent Collections

Need of Concurrent Collections

- Most of the Traditional Collections are not Thread-Safe. i.e., Multiple Threads are can access those collections simultaneously and leads to Data Inconsistency.
- Though there are some traditional Collections which are Thread Safe like Vector, Hashtable but there also only one thread can perform operation on the Collection Object. Other threads are not allowed to do any operation not even read operation. This decreases the performance of the application.
- While one Thread is iterating over the collection, no other thread is allowed to perform any update operation on the list. If we try to do so, we will get run time exception saying **ConcurrentModificationException**.
- These three problems can be overcome by using of Concurrent Collections.



```
Main.java
1+ import java.util.*;
2+ class TraditionalCollectionProblem extends Thread {
3+     static ArrayList al = new ArrayList();
4+     public void run(){
5+         try{
6+             Thread.sleep(2000);
7+         }catch (InterruptedException e){
8+             }
9+         al.add("D");
10+     }
11+     public static void main(String[] args) throws InterruptedException {
12+         al.add("A");
13+         al.add("B");
14+         al.add("C");
15+         TraditionalCollectionProblem t = new TraditionalCollectionProblem();
16+         t.start();
17+         Iterator itr = al.iterator();
18+         while(itr.hasNext()){
19+             {
20+                 String element = (String) itr.next();
21+                 System.out.println("Current Element is " + element);
22+                 Thread.sleep(3000);
23+             }
24+             System.out.println(al);
25+         }
26+     }
}
```

```
Output
java -cp /tmp/vL1bAZpjm TraditionalCollectionProblem
Current Element is A
Exception in thread "main" java.util.ConcurrentModificationException
at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1043)at java.base/java.util
.ArrayList$Itr.next(ArrayList.java:997)
at TraditionalCollectionProblem.main(TraditionalCollectionProblem.java:20)
```

What are Concurrent Collections?

- Concurrent Collections are always Thread-Safe.
- When Compared with Traditional Thread Safe Collections, performance is more because of different locking mechanism.
- While one thread iterating collection, other threads are allowed to perform update operation on collection in a safe manner. Hence, Concurrent Collections never throw ConcurrentModificationException.
- The important Concurrent Classes are:
 1. ConcurrentHashMap
 2. CopyOnWriteArrayList
 3. CopyOnWriteArraySet

10.1 ConcurrentMap

- **ConcurrentMap** is a Child Interface of **Map** Interface.
- All the Methods of Map Interface is available for ConcurrentMap Interface. Apart from that ConcurrentMap has some specific methods as well.
- There is only one Implementation Class for ConcurrentMap which is **ConcurrentHashMap**.

Methods of ConcurrentMap

- **Object putIfAbsent(Object key, Object value)**
This method will add key-value pair only if specified key is absent. If key is already there it won't do anything. This is different from normal put() method in which if key is already present then it will replace its value to the current value.
- **boolean remove(Object key, Object value)**
It removes the specified key-value pair. It is different from remove(Object key) method of traditional collection which remove entry from Map based on Key only. With that key whatever value is associated it will remove.
- **boolean replace(Object key, Object oldValue, Object newValue)**
It will replace the Value of Entry to newValue. The entry should match with key and oldValue. If match fails, nothing will happen.

10.1.1 ConcurrentHashMap



- ConcurrentHashMap is an implementation class of ConcurrentMap.
- The underlying data structure is Hashtable.
- It allows Concurrent Read Operations i.e., any number of read operations and also thread safe update operations.
- To perform read operation, no lock is required. But to perform update operation Thread requires lock but it is lock of a particular part of Map (Segment Level or Bucket Level Lock) instead of total map or total Hashtable.
- Concurrent update operation is achieved internally by dividing Map into smaller portions called Concurrency Level. The default Concurrency Level is 16.
- ConcurrentHashMap allows any number of read operations and 16 update operations at a time by default. This is because by default initial capacity of ConcurrentHashMap is 16 and default Concurrency level is 16.
- Null insertion is not allowed for both keys and values.
- While one thread is iterating, other thread can perform update operation as ConcurrentHashMap never throws ConcurrentModificationException.

Constructors of ConcurrentHashMap

- *ConcurrentHashMap m = new ConcurrentHashMap();*
Creates an empty Concurrent HashMap with default initial capacity 16 and default fill ratio 0.75 and default concurrency level of 16.
- *ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity);*
Creates an empty Concurrent HashMap with specified initial capacity and default fill ratio 0.75 and default concurrency level of 16.
- *ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio);*
Creates an empty Concurrent HashMap with specified initial capacity and specified fillRatio and default concurrency level of 16.
- *ConcurrentHashMap m = new ConcurrentHashMap(int initialCapacity, float fillRatio, int concurrencyLevel);*
Creates an empty Concurrent HashMap with specified initial capacity and specified fillRatio and also specified concurrencyLevel.

- `ConcurrentHashMap m = new ConcurrentHashMap(Map M);`
Creates an equivalent ConcurrentHashMap of specified Map M.

Example Program Snippets of ConcurrentHashMap

Main.java	Run	Output
<pre>1- import java.util.concurrent.*; 2 3 class ConcurrentHashMapDemo 4 { 5 public static void main(String[] args) 6 { 7 ConcurrentHashMap m = new ConcurrentHashMap(); 8 m.put(101,"A"); 9 m.put(102,"B"); 10 System.out.println(m); 11 m.putIfAbsent(103,"C"); 12 System.out.println(m); 13 m.putIfAbsent(101,"D"); 14 System.out.println(m); 15 m.remove(101,"D"); 16 System.out.println(m); 17 m.replace(102,"B","E"); 18 System.out.println(m); 19 } 20 }</pre>		<pre>java -cp /tmp/kcpv5jsg3N ConcurrentHashMapDemo {101=A, 102=B} {101=A, 102=B, 103=C} {101=A, 102=B, 103=C} {101=A, 102=B, 103=C} {101=A, 102=E, 103=C}</pre>
<pre>1- import java.util.concurrent.*; 2 import java.util.*; 3 class MyThread extends Thread{ 4 static ConcurrentHashMap m = new ConcurrentHashMap(); 5 public void run(){ 6 try{ 7 Thread.sleep(2000); 8 }catch(InterruptedException e){ 9 System.out.println("Child Thread Exception" + e.getMessage()); 10 } 11 m.put(103,"C"); 12 } 13 public static void main(String args[]) throws InterruptedException { 14 m.put(101,"A"); 15 m.put(102,"B"); 16 m.put(104,"D"); 17 MyThread t = new MyThread(); 18 t.start(); 19 Set s = m.keySet(); 20 Iterator itr = s.iterator(); 21 while(itr.hasNext()){ 22 Integer I1 = (Integer)itr.next(); 23 System.out.println("Main Thread Executing and Value for Current Key " + m.get(I1)); 24 Thread.sleep(3000); 25 } 26 System.out.println(m); 27 } 28 }</pre>		<pre>java -cp /tmp/kcpv5jsg3N MyThread Main Thread Executing and Value for Current Key A Main Thread Executing and Value for Current Key B Main Thread Executing and Value for Current Key C Main Thread Executing and Value for Current Key D {101=A, 102=B, 103=C, 104=D}</pre>

Difference between HashMap and ConcurrentHashMap

HashMap	ConcurrentHashMap
It is not Thread-Safe.	It is Thread-Safe.
Relatively Performance is High because Threads are not required to wait to operate on HashMap.	Relatively Performance is low because sometimes Threads are required to wait to operate on ConcurrentHashMap
While One Thread is Iterating over HashMap, other threads are not allowed to update the Map Object otherwise we will get ConcurrentModificationException .	While One Thread is Iterating over ConcurrentHashMap, other threads are allowed to update the Map Object and it won't throw ConcurrentModificationException .
Null is allowed for both keys and Values.	Null is not allowed for both keys and values else we will get NullPointerException.
Iterator of HashMap is fail-fast and it throws ConcurrentModificationException .	Iterator of ConcurrentHashMap is fail-safe and it won't throw ConcurrentModificationException .
Introduced in 1.2 Version	Introduced in 1.5 Version

Difference between ConcurrentHashMap, Hashtable and SynchronizedMap

ConcurrentHashMap	SynchronizedMap	Hashtable
We will get Thread safety without locking total map object, just with bucket level or segment level locks.	We will get thread safety by locking total map object.	We will get thread safety by locking total map object.
At a time multiple threads are allowed to perform operation on Map Object in safe manner.	At a time only one thread is allowed to perform operation on Map Object.	At a time only one thread is allowed to perform operation on Hashtable Object.
Read operation can be performed without any lock while write operation can be performed by using bucket level or Segmentation level locks.	Every read and Write Operation require a total map Object Lock.	Every read and Write Operation require a total Hashtable Object Lock.
While one thread is iterating Map Object the other threads are allowed to modify map object and we won't get ConcurrentModificationException .	While one thread is iterating Map Object the other threads are not allowed to modify map object and we will get ConcurrentModificationException .	While one thread is iterating Hashtable Object the other threads are not allowed to modify Hashtable object and we will get ConcurrentModificationException .
Iterator is Fail-Safe.	Iterator is Fail-Fast.	Iterator is Fail-Fast.
Null Key and Null Values are not allowed.	Null Key and Null Values are allowed.	Null Key and Null Values are allowed.
Introduced in 1.5 Version	Introduced in 1.2 Version	Introduced in 1.0 Version.

10.2 CopyOnWriteArrayList

- It is a Thread-Safe version of ArrayList as the name indicates CopyOnWriteArrayList creates a cloned copy of underlying ArrayList Object for every update operation at certain point both will synchronize automatically which is taken care by JVM internally.
- As Update operation is performed on cloned copy, there is no effect for the threads which perform read operation.
- It is costly to use because for every update operation a cloned copy will be created. Hence CopyOnWriteArrayList is a best choice if several read operations and very few number of write operations are required to perform.
- Insertion Order is preserved. Duplicates, Heterogenous, Null objects are allowed to insert.
- It implements Serializable, Clonable, and RandomAccess Interface.
- While one thread is iterating CopyOnWriteArrayList Object the other threads are allowed to modify object and we won't get ConcurrentModificationException. i.e., Iterator is fail-safe.
- Iterator of ArrayList can perform remove() operation while Iterator of CopyOnWriteArrayList can't perform remove() operation otherwise we will get UnsupportedOperationException.

Constructors of CopyOnWriteArrayList

1. `CopyOnWriteArrayList c = new CopyOnWriteArrayList();`
2. `CopyOnWriteArrayList c = new CopyOnWriteArrayList(Collection C);`
3. `CopyOnWriteArrayList c = new CopyOnWriteArrayList(Object[] O);`

Methods of CopyOnWriteArrayList

- **boolean addIfAbsent(Object O)**
Add Object O if it is not present in a List else it won't perform any operation.
- **int addAllAbsent(Collection C)**
The elements of Collection C will be added to the list if elements are absent and return the number of elements added.

Example Program Snippet of CopyOnWriteArrayList

The first screenshot shows a Java program named `Main.java` that demonstrates the `CopyOnWriteArrayList` class. It imports `java.util.concurrent.*` and `java.util.*`. The `main` method creates two `CopyOnWriteArrayList` objects, `c1` and `c2`. `c1` is initialized with `A` and `B`, and `c2` is initialized with `A` and `E`. The program then prints the contents of both lists and the final state of `c1` after adding `c2`.

```
1- import java.util.concurrent.*;
2 import java.util.*;
3- class CopyOnArrayListDemo {
4     public static void main(String args[]) throws InterruptedException
5     {
6         ArrayList l1 = new ArrayList();
7         l1.add("A");
8         l1.add("B");
9         System.out.println("ArrayList 1 " +l1);
10        CopyOnWriteArrayList c1 = new CopyOnWriteArrayList();
11        c1.addIfAbsent("A");
12        c1.addIfAbsent("C");
13        c1.addAll(l1);
14        System.out.println("Concurrent ArrayList " +c1);
15        ArrayList l2 = new ArrayList();
16        l2.add("A");
17        l2.add("E");
18        System.out.println("ArrayList 2 " +l2);
19        c1.addAllAbsent(l2);
20        System.out.println("Concurrent ArrayList Final " +c1);
21    }
22 }
```

The output of the first program is:

```
java -cp /tmp/kcpv5jsg3N CopyOnArrayListDemo
ArrayList 1 [A, B]
Concurrent ArrayList [A, C, A, B]
ArrayList 2 [A, E]
Concurrent ArrayList Final [A, C, A, B, E]
```

The second screenshot shows a Java program named `Main.java` that demonstrates the `CopyOnWriteArrayList` class in a concurrent context. It imports `java.util.concurrent.*` and `java.util.*`. The `main` method creates a `CopyOnWriteArrayList` object `m` and a `MyThread` object `t`. `m` is initialized with `101`, `102`, `104`, and `105`. `t` is started and it adds `103` to `m`. The program then prints the contents of `m` and the final state of `m` after adding `103`.

```
1- import java.util.concurrent.*;
2 import java.util.*;
3- class MyThread extends Thread{
4     static CopyOnWriteArrayList m = new CopyOnWriteArrayList();
5     public void run(){
6         try{
7             Thread.sleep(2000);
8         }catch(InterruptedException e){
9             System.out.println("Child Thread Exception" + e.getMessage());
10        }
11        m.add(103);
12    }
13- public static void main(String args[]) throws InterruptedException {
14        m.add(101);
15        m.add(102);
16        m.add(104);
17        m.add(105);
18        MyThread t = new MyThread();
19        t.start();
20        Iterator itr = m.iterator();
21-        while(itr.hasNext()){
22            Integer i1 = (Integer)itr.next();
23            System.out.println("Main Thread Executing and Current Value " + i1);
24            Thread.sleep(3000);
25        }
26        System.out.println(m);
27    }
28 }
```

The output of the second program is:

```
java -cp /tmp/kcpv5jsg3N MyThread
Main Thread Executing and Current Value 101
Main Thread Executing and Current Value 102
Main Thread Executing and Current Value 104
Main Thread Executing and Current Value 105
[101, 102, 104, 105, 103]
```

If we replace `CopyOnWriteArrayList` here with `Normal ArrayList` then we will get exception `ConcurrentModificationException`.

```
Main.java
1- import java.util.concurrent.*;
2- import java.util.*;
3- class MyThread extends Thread{
4-     static ArrayList m = new ArrayList();
5-     public void run(){
6-         try{
7-             Thread.sleep(2000);
8-         }catch (InterruptedException e){
9-             System.out.println("Child Thread Exception" + e.getMessage());
10        }
11        m.add(103);
12    }
13- public static void main(String args[]) throws InterruptedException {
14    m.add(101);
15    m.add(102);
16    m.add(104);
17    m.add(105);
18    MyThread t = new MyThread();
19    t.start();
20    Iterator itr = m.iterator();
21-    while(itr.hasNext()){
22        Integer I1 = (Integer)itr.next();
23        System.out.println("Main Thread Executing and Current Value " + I1);
24        Thread.sleep(3000);
25    }
26    System.out.println(m);
27 }
28 }
```

```
Output
- java -cp /tmp/kcpv5jsg3N MyThread
Main Thread Executing and Current Value 101Exception in thread "main" java.util
.ConcurrentModificationException
at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1043)
at java.base/java.util.ArrayList$Itr.next(ArrayList.java:997)
at MyThread.main(MyThread.java:22)
```

We cannot perform remove operation on CopyOnWriteArrayList. If we do so we will get UnsupportedOperationException.

```
Main.java
1- import java.util.concurrent.*;
2- import java.util.*;
3- class RemoveDemo{
4-     public static void main(String args[])
5-     {
6-         CopyOnWriteArrayList m = new CopyOnWriteArrayList();
7-         m.add(101);
8-         m.add(102);
9-         m.add(103);
10        m.add(104);
11        m.add(105);
12        m.add(106);
13        Iterator itr = m.iterator();
14-        while(itr.hasNext()){
15            Integer I1 = (Integer)itr.next();
16            if(I1 == 104)
17            {
18                itr.remove();
19            }
20        }
21        System.out.println(m);
22    }
23 }
```

```
Output
- java -cp /tmp/kcpv5jsg3N RemoveDemo
Exception in thread "main" java.lang.UnsupportedOperationException
at java.base/java.util.concurrent.CopyOnWriteArrayList$COWIterator.remove(CopyOnWriteArrayList.java:1124)
at RemoveDemo.main(RemoveDemo.java:18)
```

But we can perform remove operation on Normal ArrayList.

```
Main.java
1- import java.util.concurrent.*;
2- import java.util.*;
3- class RemoveDemo{
4-     public static void main(String args[])
5-     {
6-         ArrayList m = new ArrayList();
7-         m.add(101);
8-         m.add(102);
9-         m.add(103);
10        m.add(104);
11        m.add(105);
12        m.add(106);
13        Iterator itr = m.iterator();
14-        while(itr.hasNext()){
15            Integer I1 = (Integer)itr.next();
16            if(I1 == 104)
17            {
18                itr.remove();
19            }
20        }
21        System.out.println(m);
22    }
23 }
```

```
Output
- java -cp /tmp/kcpv5jsg3N RemoveDemo
[101, 102, 103, 105, 106]
```

Every update operation will be performed on a separate clone copy. Hence, after getting iterator if we are trying to perform any modification to the list, then it won't get reflected on the iterator.

But if we replace `CopyOnWriteArrayList` with Normal `ArrayList` then we will get exception.

Difference between ArrayList and CopyOnWriteArrayList

ArrayList	CopyOnWriteArrayList
Not thread Safe	Thread Safe
Iterator can perform both read and remove operation.	Iterator can perform only read operation but not remove operation. If we try to do it will throw UnsupportedOperationException.
Iterator is Fail-Fast.	Iterator is Fail-Safe.
While one Thread is iterating List Object, then other threads are not allowed to perform any modification on that list and throw ConcurrentModificationException.	While one Thread is iterating List Object, then other threads are allowed to perform modification on that list and won't throw ConcurrentModificationException.

Difference between CopyOnWriteArrayList, SynchronizedList and Vector

CopyOnWriteArrayList	SynchronizedList	Vector
We will get Thread safety because every update will be performed on separate cloned copy.	We will get thread safety because at a time List can be accessed by only one Thread.	We will get thread safety because at a time Vector can be accessed by only one Thread.
At a time multiple threads are allowed to perform operation.	At a time only one thread is allowed to perform operation on List Object.	At a time only one thread is allowed to perform operation on Vector Object.
While one thread is iterating List Object the other threads are allowed to modify List object and we won't get ConcurrentModificationException .	While one thread is iterating List Object the other threads are not allowed to modify List object and we will get ConcurrentModificationException .	While one thread is iterating Vector Object the other threads are not allowed to modify Vector object and we will get ConcurrentModificationException .
Iterator is Fail-Safe.	Iterator is Fail-Fast.	Iterator is Fail-Fast.
Iterator can perform only read operation but not remove operation. If we try to do it will throw UnsupportedOperationException .	Iterator can perform both read and remove operation.	Iterator can perform both read and remove operation.
Introduced in 1.5 Version	Introduced in 1.2 Version	Introduced in 1.0 Version.

10.3 CopyOnWriteArraySet

- It is a Thread Safe Version of Set.
- Internally implemented by CopyOnWriteArrayList.
- Insertion Order is preserved, duplicates are not allowed, Heterogenous Objects and null insertions is possible.
- Multiple Threads can able to perform Read Operation Simultaneously but for every update operation a separate cloned copy will be created.
- It is costly to use because for every update operation a cloned copy will be created. Hence CopyOnWriteArraySet is a best choice if several read operations and very few number of write operations are required to perform.
- While one thread is iterating CopyOnWriteArraySet Object the other threads are allowed to modify object and we won't get ConcurrentModificationException. i.e., Iterator is fail-safe.
- Iterator of ArrayList can perform remove() operation while Iterator of CopyOnWriteArraySet can't perform remove() operation otherwise we will get UnsupportedOperationException.

Constructor of CopyOnWriteArraySet

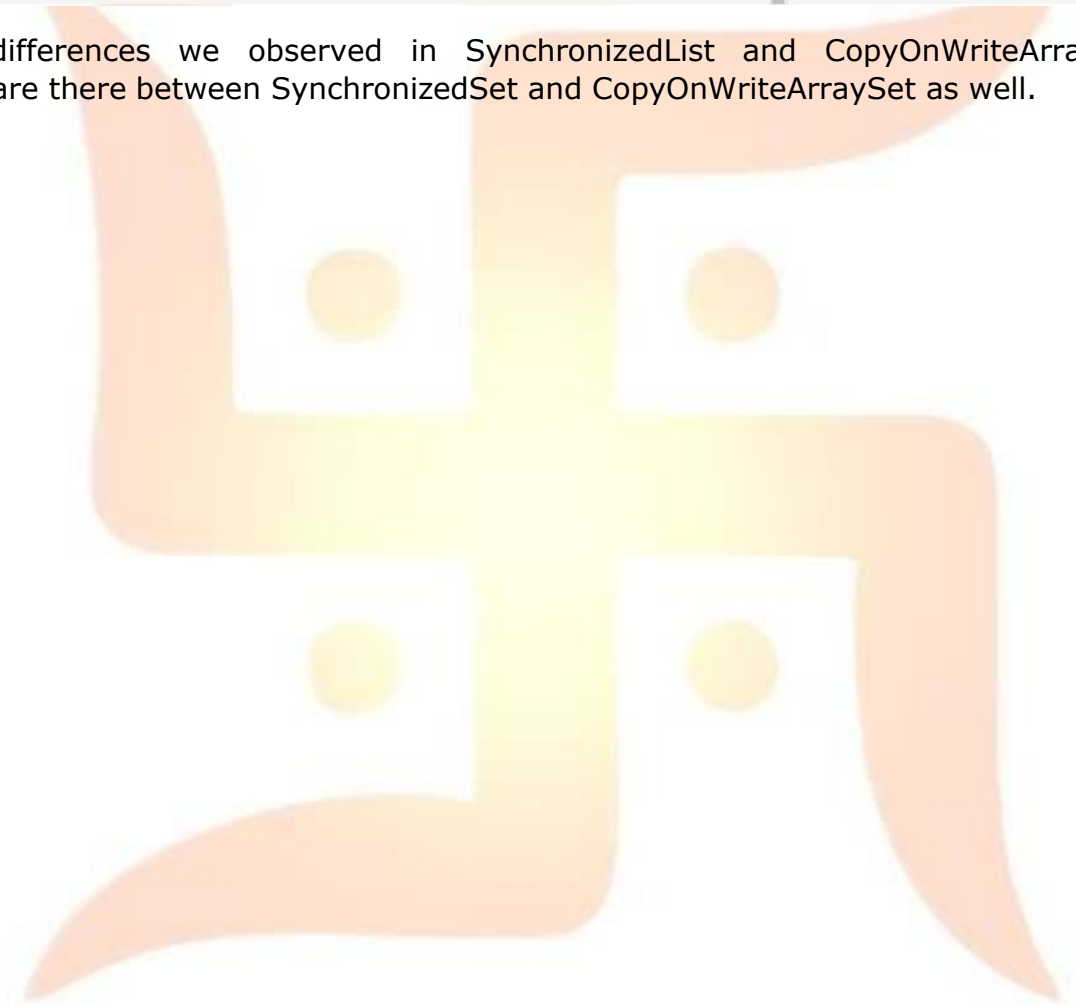
1. *CopyOnWriteArraySet S = new CopyOnWriteArraySet();*
2. *CopyOnWriteArraySet S = new CopyOnWriteArraySet(Collection C);*

There is no new method defined for CopyOnWriteArraySet class. Whatever the methods Collection and Set Interface contains, same method CopyOnWriteArraySet also has.

```
Main.java
1- import java.util.concurrent.*;
2- import java.util.*;
3- class CopyOnWriteArraySetDemo
4- {
5-     public static void main(String args[])
6-     {
7-         CopyOnWriteArraySet m = new CopyOnWriteArraySet();
8-         m.add(101);
9-         m.add(102);
10-        m.add(103);
11-        m.add("Vikash");
12-        m.add(104);
13-        m.add(105);
14-        m.add(null);
15-        m.add(106);
16-        m.add(107);
17-        m.add(102);
18-        System.out.println(m);
19-    }
20- }
```

```
Output
java -cp /tmp/kcpv5jsg3N CopyOnWriteArraySetDemo
[101, 102, 103, Vikash, 104, 105, null, 106, 107]
```

Whatever differences we observed in SynchronizedList and CopyOnWriteArrayList same differences are there between SynchronizedSet and CopyOnWriteArraySet as well.



11.. Generics

The main objective of Generics is to provide Type Safety and to resolve Type Casting Problem.

11.1 Type-Safety

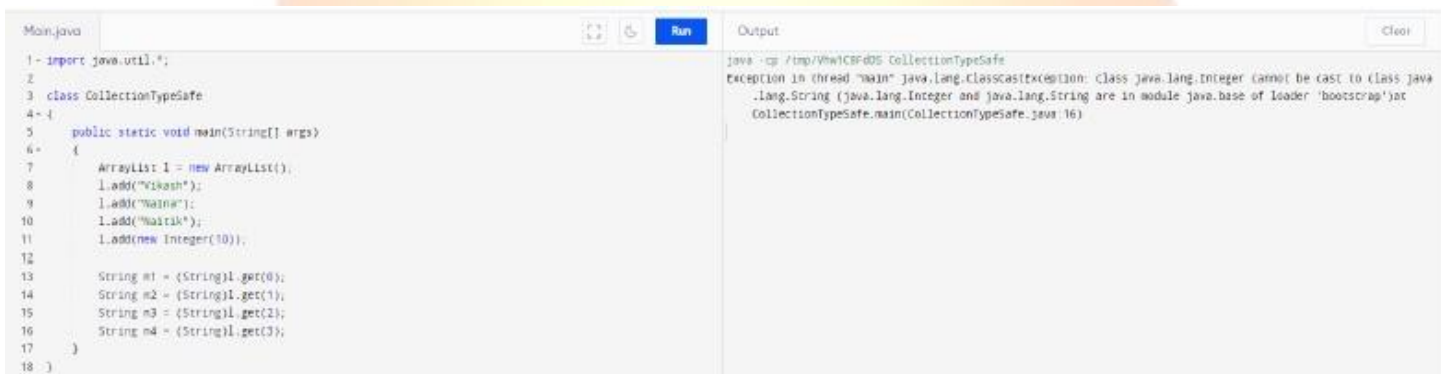
- Arrays are Type-Safe. i.e., We can give guarantee for the Type of Objects present in an Array. For example, if our program requirement is to hold the only String type Objects, we can choose String array. If mistakenly, we try to add any other object then we will get compile time error i.e., incompatible Types.



```
Main.java
1 class TypeSafe
2 {
3     public static void main(String args[])
4     {
5         String[] s = new String[10];
6         s[0] = "Vikash";
7         s[1] = "Akash";
8         s[2] = new Integer(10);
9     }
10 }
```

```
Output
ERROR!
javac /tmp/kcpv5jsg3k/TypeSafe.java
/tmp/kcpv5jsg3k/TypeSafe.java:8: error: incompatible types: Integer cannot be converted to String
    s[2] = new Integer(10);
    ^
Note: /tmp/kcpv5jsg3k/TypeSafe.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
1 error
```

- Here in above program we see String Array can contain only String type of Objects which provides type safety.
- But if we talk about collections, they are not type-safe. As collection allows us to store Heterogenous Objects to store which will not throw any error at compile time but if we try to get those values in particular data type then it might throw run time exception saying ClassCastException. In below example our requirement is to hold only String object in ArrayList but mistakenly I have added Integer value which is not giving any compile time error but throwing run time exception as we are expecting specific type data.



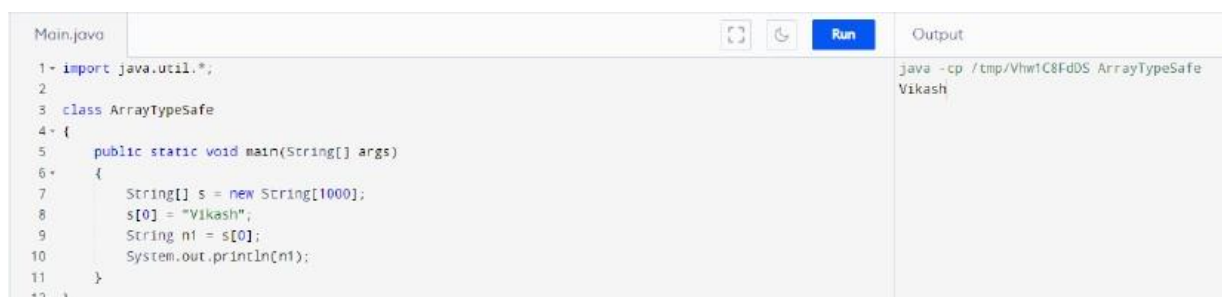
```
Main.java
1 import java.util.*;
2
3 class CollectionTypeSafe
4 {
5     public static void main(String[] args)
6     {
7         ArrayList l = new ArrayList();
8         l.add("Vikash");
9         l.add("Wainu");
10        l.add("Waltik");
11        l.add(new Integer(10));
12
13        String m1 = (String)l.get(0);
14        String m2 = (String)l.get(1);
15        String m3 = (String)l.get(2);
16        String m4 = (String)l.get(3);
17    }
18 }
```

```
Output
java -cp /tmp/Vhw1C8Fd0S CollectionTypeSafe
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')
CollectionTypeSafe.main(CollectionTypeSafe.java:16)
```

- So, Generics is a Concept which is added to provide Type Safety to Collections.

11.2 Type Casting

- In case of Arrays, at the time of Retrieval data, type-casting is not required. Because there is always a guarantee that in Array which we have declared of Particular type it will contain those type data only. For Example – If we have String[] array then it is guarantee that String[] array will contain only String Objects.



```
Main.java
1 import java.util.*;
2
3 class ArrayTypeSafe
4 {
5     public static void main(String[] args)
6     {
7         String[] s = new String[1000];
8         s[0] = "Vikash";
9         String n1 = s[0];
10        System.out.println(n1);
11    }
12 }
```

```
Output
java -cp /tmp/Vhw1C8Fd0S ArrayTypeSafe
Vikash
```


- But in case of Collection, which can store any objects i.e., Heterogenous Objects, while retrieval we cannot guarantee that the data store in a collection is of particular type or not so we need to do compulsory Type Casting in case of Collection.

Main.java	Output
<pre> 1- import java.util.*; 2 3 class CollectionTypeSafe 4 { 5 public static void main(String[] args) 6 { 7 ArrayList l = new ArrayList(); 8 l.add("Vikash"); 9 10 String n1 = l.get(0); 11 System.out.println(n1); 12 } 13 } </pre>	<pre> ERROR! javac /tmp/Vhw1C8FdDS/CollectionTypeSafe.java /tmp/Vhw1C8FdDS/CollectionTypeSafe.java:10: error: incompatible types: Object cannot be converted to String String n1 = l.get(0); ^ Note: /tmp/Vhw1C8FdDS/CollectionTypeSafe.java uses unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details. 1 error </pre>

Main.java	Output
<pre> 1- import java.util.*; 2 3 class CollectionTypeSafe 4 { 5 public static void main(String[] args) 6 { 7 ArrayList l = new ArrayList(); 8 l.add("Vikash"); 9 10 String n1 = (String)l.get(0); 11 System.out.println(n1); 12 } 13 } </pre>	<pre> java -cp /tmp/Vhw1C8FdDS CollectionTypeSafe Vikash </pre>

- To Resolve this Type casting problem also we need Generics Concept.
- To Resolve this Type Safety and Type Casting Problem in Collection, Generics got introduced in Java Version 1.5.

How can we use Generics in Collection?

- To Hold only String type of Objects in a Collection say ArrayList we can write the Generic Version of ArrayList as:
`ArrayList<String> al = new ArrayList<String>();`

Now, we can add only String Objects to ArrayList al. If we try to add any other objects say Integer, then it will throw Compile Time error – Incompatible Type. Hence, Generic version of ArrayList provides Type Safety. And also, in Generic Version of ArrayList we are not required to do any Type casting while retrieving data.

Example:

Main.java	Output
<pre> 1- import java.util.*; 2 3 class CollectionTypeSafe 4 { 5 public static void main(String[] args) 6 { 7 ArrayList<String> l = new ArrayList<String>(); 8 l.add("Vikash"); 9 String n1 = l.get(0); 10 System.out.println(n1); 11 } 12 } </pre>	<pre> java -cp /tmp/Vhw1C8FdDS CollectionTypeSafe Vikash </pre>

```
Main.java
1- import java.util.*;
2
3 class CollectionTypeSafe {
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> l = new ArrayList<String>();
8         l.add("Vijash");
9         l.add(new Integer(20));
10        String n1 = l.get(0);
11        System.out.println(n1);
12    }
13 }
```

```
Output
ERROR!
javac /tmp/Vhw1C6FdDS/CollectionTypeSafe.java
/tmp/Vhw1C6FdDS/CollectionTypeSafe.java:9: error: incompatible types: Integer cannot be converted to String
    l.add(new Integer(20));
      ^
Note: /tmp/Vhw1C6FdDS/CollectionTypeSafe.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

In this way, we can create Generic version of any Collection Class by Using Syntax:

Collection<T>
Map<K,V>

Here Collection or Map is a Base Type while <T> or <K,V> is a parameter type. Polymorphism concept can be used for Base Type but no Polymorphism can be used for parameter type.

Hence all the below are possible cases of Polymorphism.

```
Main.java
1- import java.util.*;
2- class GenericDemo {
3     public static void main(String[] args)
4     {
5         ArrayList<String> l1 = new ArrayList<String>();
6         List<String> l2 = new ArrayList<String>();
7         Collection<String> l3 = new ArrayList<String>();
8     }
9 }
10
```

```
Output
java -cp /tmp/z47Cx3RhG0 GenericDemo
```

But we cannot change Parameter type String to Object. It will give Compile Time Error:

```
Main.java
1- import java.util.*;
2- class GenericDemo {
3     public static void main(String[] args)
4     {
5         ArrayList<Object> l1 = new ArrayList<String>();
6     }
7 }
```

```
Output
ERROR!
javac /tmp/z47Cx3RhG0/GenericDemo.java
/tmp/z47Cx3RhG0/GenericDemo.java:5: error: incompatible types: ArrayList<String> cannot be converted to
    ArrayList<Object> l1 = new ArrayList<String>();
                                ^
1 error
```

For the Parameter type, we can take any class or Interface Name but we cannot use primitive data type as a Parameter Type.

```
Main.java
1- import java.util.*;
2- class GenericDemo {
3     public static void main(String[] args)
4     {
5         ArrayList<int> l1 = new ArrayList<int>();
6     }
7 }
8
9
10
11
12
13
```

```
Output
ERROR!
javac /tmp/z47Cx3RhG0/GenericDemo.java
/tmp/z47Cx3RhG0/GenericDemo.java:5: error: unexpected type
    ArrayList<int> l1 = new ArrayList<int>();
                                ^
    required: reference
    found:    int
/tmp/z47Cx3RhG0/GenericDemo.java:5: error: unexpected type
    ArrayList<int> l1 = new ArrayList<int>();
                                ^
    required: reference
    found:    int
2 errors
```

11.3 Generic Classes

- Until 1.4 Version, normal non-generic version of ArrayList class was declared as follows:

```

1 class ArrayList
2 {
3     add(Object O){
4
5     }
6
7     Object get(int index){
8
9     }
10 }

```

Since the argument for add method is Object O. Hence, we can add any type of Object in an ArrayList which is not making ArrayList as Type-Safe. The return type of get() method is Object, hence at the time of retrieval we have to perform type-casting.

- From 1.5 version, a generic version of ArrayList Class is declared as follows:

```

class ArrayList<T> //T is a Type Parameter
{
    add(T t){

    }

    T get(int index){

    }
}

```

Now, during run time based on our requirement T will be replaced with our Provided type. Now, to hold only String Objects, generic version of ArrayList can be created as Follows:
ArrayList<String> al = new ArrayList<String>();

For this requirement, Compiler considered version of ArrayList is as below:

```

class ArrayList<String>
{
    add(String t){

    }

    String get(int index){

    }
}

```

Since Argument of add method is String type. We can add only String Objects to ArrayList now which provides Type-Safety. If we try to add any other Object it will throw compile time error.

Main.java	Output
<pre> 1- import java.util.*; 2- class GenericDemo { 3 public static void main(String[] args) 4 { 5 ArrayList<String> l1 = new ArrayList<String>(); 6 l1.add(new Integer(10)); 7 } 8 } 9 </pre>	<pre> ERROR! javac /tmp/z47Cx3RhG0/GenericDemo.java /tmp/z47Cx3RhG0/GenericDemo.java:6: error: incompatible types: Integer cannot be converted to String l1.add(new Integer(10)); ^ Note: /tmp/z47Cx3RhG0/GenericDemo.java uses or overrides a deprecated API. Note: Recompile with -Xlint:Deprecation for details. Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output 1 error </pre>

Also, since return type of Get Method is Object type so we are not required to perform any type casting.

Main.java	Run	Output
<pre>1 import java.util.*; 2 class GenericDemo { 3 public static void main(String[] args) 4 { 5 ArrayList<String> l1 = new ArrayList<String>(); 6 l1.add("Vikash"); 7 String n = l1.get(0); 8 System.out.println(n); 9 } 10 }</pre>		<pre>java -cp /tmp/z47Cx3RhGO GenericDemo Vikash</pre>

In generics, we are associated type parameter to the class. Such type of parameterized classes are nothing but Generic Classes or Template Classes.

Based on requirement, we can define our own Generic Classes also. Example:

```
1 class Account<T>
2 {
3
4 }
5 class Demo{
6     public static void main(String args[]){
7         Account<Gold> a1 = new Account<Gold>();
8         Account<Silver> a2 = new Account<Silver>();
9     }
10 }
```

Main.java	Run	Output
<pre>1 class Gen<T> { 2 T ob; 3 Gen(T ob){ 4 this.ob = ob; 5 } 6 public void show(){ 7 System.out.println("The type of ob : " + ob.getClass().getName()); 8 } 9 public T getOb(){ 10 return ob; 11 } 12 } 13 class Demo { 14 public static void main(String args[]){ 15 Gen<String> g1 = new Gen<String>("Vikash"); 16 g1.show(); 17 Gen<Integer> g2 = new Gen<Integer>(10); 18 g2.show(); 19 } 20 }</pre>		<pre>java -cp /tmp/z47Cx3RhGO Demo The type of ob : java.lang.String The type of ob : java.lang.Integer</pre>

Note: While creating Generic classes we use ClassName<T>. We can replace this T with any valid Java Identifier without any issue. But its recommended to use letter T as T denotes Type Parameter.

Also based on our requirement we can declare any number of Type Parameter example ClassName<A,B,C>. Common example of this class is HashMap<K,V>.

11.4 Bounded Types

We can bound the type parameter for a particular range by using extends keyword. Such types are called Bounded Types. We cannot use implements or super keyword to create Bounded Types but we can replace implements keyword purpose with extends keyword.

```
1 class Gen<T> {
2     T ob;
3     Gen(T ob){
4         this.ob = ob;
5     }
6     public void show(){
7         System.out.println("The type of ob : " + ob.getClass().getName());
8     }
9     public T getOb(){
10        return ob;
11    }
12 }
13 class Demo {
14     public static void main(String args[]){
15         Gen<String> g1 = new Gen<String>("Vikash");
16         g1.show();
17         Gen<Integer> g2 = new Gen<Integer>(10);
18         g2.show();
19     }
20 }
```

Here class Gen is a Generic Class where We can replace T with any Object type say String, Student, Employee etc. i.e., it is an unbounded Generic class. We have used this unbounded Generic class with String and Integer Object in above program. But we can bound this class to except only Number Objects by making a below Change:

```
class Gen<T extends Number> {
    T ob;
    Gen(T ob){
        this.ob = ob;
    }
    public void show(){
        System.out.println("The type of ob : " + ob.getClass().getName());
    }
    public T getOb(){
        return ob;
    }
}
```

Syntax of Bounded Type

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

Here X is any class or Interface. If X is a class, then as a type parameter, we can either pass X or its child classes. If X is an interface, then as a type parameter, we can either pass X or its implementation classes.

Main.java	Output
<pre> 1- class Gen<T extends Number> { 2 T ob; 3- Gen(T ob){ 4 this.ob = ob; 5 } 6- public void show(){ 7 System.out.println("The type of ob : " + ob.getClass().getName()); 8 } 9- public T getOb(){ 10 return ob; 11 } 12 } 13- class Demo { 14- public static void main(String args[]){ 15 Gen<String> g1 = new Gen<String>("Vikash"); 16 g1.show(); 17 Gen<Integer> g2 = new Gen<Integer>(10); 18 g2.show(); 19 } 20 } </pre>	<pre> ERROR! javac /tmp/247Cx3RhG0/Demo.java /tmp/247Cx3RhG0/Demo.java:15: error: type argument String is not within bounds of type-variable T Gen<String> g1 = new Gen<String>("Vikash"); ^ where T is a type-variable: T extends Number declared in class Gen /tmp/247Cx3RhG0/Demo.java:17: error: type argument Integer is not within bounds of type-variable T Gen<Integer> g2 = new Gen<Integer>(10); ^ where T is a type-variable: T extends Number declared in class Gen 2 errors </pre>

Here, in above program, since Gen is a Type Bounded Generic class where we can use either Number class or its Subclass. But, String is not a child class of Number so it will give compile time error.

We can define bounded types even in combination also.

Example:

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

Here, Test is a bounded generic class where we can pass either Number class or its child class and the implementation class of Runnable.

Some rules:

- Type Parameter cannot extend two classes as Java doesn't support Multiple Inheritance but can extend two interfaces. For example:

class Test<T extends Number & Thread> {}

Invalid as Number and Thread both are classes.

class Test<T extends Runnable & Comparable> {}

Valid as Runnable and Comparable both are interfaces.

- After extends first will always be a class and remaining can be one or more interface. For example:

class Test<T extends Number & Runnable & Comparable> {}

Valid as T extends only one class which is Number and two interfaces Runnable and Comparable.

class Test<T extends Runnable & Number> {}

Invalid as first cannot be Interface. As in Java first we extend class then only implement interface.

11.5 Generic Methods and Wild Card Character (?)

- Use Case 1**

```
public void m1(ArrayList<String> al)
{
}
}
```

We can call this method by passing an ArrayList of only String Type. If we try to Pass an ArrayList of any other type say Integer, Student etc. then in that case we have to declare

other methods with `ArrayList<Integer>` or `ArrayList<Student>` as parameter. Above method won't work in this case. So,

```
ArrayList<String> l1 = new ArrayList<String>();  
ArrayList<Integer> l2 = new ArrayList<Integer>();  
m1(l1); is valid.  
m1(l2); is Invalid.
```

- **Use Case 2**

```
public void m1(ArrayList<?> al)  
{  
}
```

We can call this method by passing an `ArrayList` of any type. We can pass here `String`, `Integer`, `Student`, `Employee` etc. based on our requirement. But within a method `m1` we can't add anything to the List except null because we don't know the type exactly. Null is allowed because it is valid value for any type. This type of methods are best suitable for read only operation but while performing write operation it is of no use. So,

```
ArrayList<String> l1 = new ArrayList<String>();  
ArrayList<Integer> l2 = new ArrayList<Integer>();  
m1(l1); is valid.  
m1(l2); is valid.
```

- **Use Case 3**

```
public void m1(ArrayList<? extends X> al)  
{  
}
```

We can call this method by passing an `ArrayList` of `X` type and its child classes if `X` is a class and also, we can call this method by passing the `X` type or the implementation class of `X` if `X` is an interface. But within a method `m1` we can't add anything to the List except null because we don't know the type exactly. Null is allowed because it is valid value for any type. This type of methods is also best suitable for read only operation.

- **Use Case 4**

```
public void m1(ArrayList<? super X> al)  
{  
}
```

We can call this method by passing an `ArrayList` of `X` type and its super classes if `X` is a class and also, we can call this method by passing the `X` type or the super class of implementation class of `X` if `X` is an interface. But within a method `m1` we can add only `X` type to the List apart from null.

Now:

1. `ArrayList<String> al = new ArrayList<String>();` Valid
2. `ArrayList<?> al = new ArrayList<String>();` Valid
3. `ArrayList<?> al = new ArrayList<Integer>();` Valid
4. `ArrayList<? extends Number> al = new ArrayList<Integer>();` Valid
5. `ArrayList<? extends Number> al = new ArrayList<String>();` Invalid

Main.java	Output
<pre> 1- import java.util.*; 2- class GenericsDemo { 3- public static void main(String[] args) { 4- ArrayList<? extends Number> al = new ArrayList<String>(); 5- } 6- } 7- 8- </pre>	<pre> ERROR! javac /tmp/j1lOuX1D9k/GenericsDemo.java /tmp/j1lOuX1D9k/GenericsDemo.java:4: error: incompatible types: ArrayList<String> cannot be converted to ArrayList<? extends Number> ^ ArrayList<? extends Number> al = new ArrayList<String>(); ^ 1 error </pre>

6. `ArrayList<? super String> al = new ArrayList<Object>();` Valid

7. `ArrayList<?> al = new ArrayList<?>();` Invalid

Main.java	Output
<pre> 1- import java.util.*; 2- class GenericsDemo { 3- public static void main(String[] args) { 4- ArrayList<?> al = new ArrayList<?>(); 5- } 6- } 7- 8- </pre>	<pre> ERROR! javac /tmp/j1lOuX1D9k/GenericsDemo.java /tmp/j1lOuX1D9k/GenericsDemo.java:4: error: unexpected type ArrayList<?> al = new ArrayList<?>(); ^ required: class or interface without bounds found: ? 1 error </pre>

8. `ArrayList<?> al = new ArrayList<? extends Number>();` Invalid

Main.java	Output
<pre> 1- import java.util.*; 2- class GenericsDemo { 3- public static void main(String[] args) { 4- ArrayList<?> al = new ArrayList<? extends Number>(); 5- } 6- } 7- 8- </pre>	<pre> ERROR! javac /tmp/j1lOuX1D9k/GenericsDemo.java /tmp/j1lOuX1D9k/GenericsDemo.java:4: error: unexpected type ArrayList<?> al = new ArrayList<? extends Number>(); ^ required: class or interface without bounds found: ? extends Number 1 error </pre>

11.6 Generic Methods

- We can declare type parameter either at Class level or at Method level.
- Declaring type parameter at Class level:

```
class GenDemo<T>{}
```

We can use this T anywhere inside this class based on our requirement.
- Declaring type parameter at Method Level:

```
public <T>void method1(T ob){}
```

We can use T anywhere inside this method based on our requirement.
- Just like Generic Class we can define Bounded types at Method Level also which follows the same rules as that of class level.

Communication with Non-Generic Code

- If we send Generic Object to Non-Generic Area then it starts behaving like Non-Generic Object. Similarly, if we send Non-Generic Object to Generic Area then it starts behaving like Generic Object. So, based on the location at which Object is present it's behavior will be defined.

Examples:

Behavior of Generic Object in Non-Generic Area.

Main.java	Output
<pre>1- import java.util.*; 2- class GenericsCommunicationDemo { 3 public static void m1(ArrayList al) 4 { 5 //Non Generic Area ArrayList al behave as a Non-generic Object 6 al.add(10); 7 al.add(10.5); 8 al.add(true); 9 al.add("Vishwakarma"); 10 } 11 public static void main(String[] args) 12 { 13 //Generic Area ArrayList al behave as a generic Object 14 ArrayList<String> al = new ArrayList<String>(); 15 al.add("Naina"); 16 al.add("Vikash"); 17 al.add("Akash"); 18 m1(al); 19 System.out.println(al); 20 } 21 }</pre>	<pre>java -cp /tmp/ij10uX109k GenericsCommunicationDemo [Naina, Vikash, Akash, 10, 10.5, true, Vishwakarma]</pre>

Main.java	Output
<pre>1- import java.util.*; 2- class GenericsCommunicationDemo { 3 public static void m1(ArrayList al) 4 { 5 //Non Generic Area ArrayList al behave as a Non-generic Object 6 al.add(10); 7 al.add(10.5); 8 al.add(true); 9 al.add("Vishwakarma"); 10 } 11 public static void main(String[] args) 12 { 13 //Generic Area ArrayList al behave as a generic Object 14 ArrayList<String> al = new ArrayList<String>(); 15 al.add("Naina"); 16 al.add("Vikash"); 17 al.add("Akash"); 18 m1(al); 19 System.out.println(al); 20 al.add(10); //Compile Time Error 21 } 22 }</pre>	<pre>ERROR! javac /tmp/ij10uX109k/GenericsCommunicationDemo.java /tmp/ij10uX109k/GenericsCommunicationDemo.java:20: error: incompatible types: int cannot be converted to String al.add(10); //Compile Time Error ^ Note: /tmp/ij10uX109k/GenericsCommunicationDemo.java uses unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details. Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output 1 error</pre>

Behavior of Non-Generic Object in Generic Area.

Main.java	Output
<pre>1- import java.util.*; 2- class GenericsCommunicationDemo { 3 public static void m1(ArrayList<Number> al) 4 { 5 //Generic Area ArrayList al behave as a generic Object 6 al.add(10); 7 al.add(20); 8 al.add(30); 9 } 10 public static void main(String[] args) 11 { 12 //Non Generic Area ArrayList al behave as a non-generic Object 13 ArrayList al = new ArrayList(); 14 al.add("Naina"); 15 al.add("Vikash"); 16 al.add("Akash"); 17 al.add(10); 18 al.add(10.5); 19 al.add(true); 20 m1(al); 21 System.out.println(al); 22 al.add(10); 23 } 24 }</pre>	<pre>java -cp /tmp/ij10uX109k GenericsCommunicationDemo [Naina, Vikash, Akash, 10, 10.5, true, 10, 20, 30]</pre>

Main.java	Output
<pre>1- import java.util.*; 2- class GenericsCommunicationDemo { 3- public static void m1(ArrayList<Number> al) 4- { 5- //Generic Area ArrayList al behave as a generic Object 6- al.add(10); 7- al.add(20); 8- al.add(30); 9- al.add("Vikash"); // Compile time Error 10- } 11- public static void main(String[] args) 12- { 13- //Non Generic Area ArrayList al behave as a non-generic Object 14- ArrayList al = new ArrayList(); 15- al.add("Naina"); 16- al.add("Vikash"); 17- al.add("Akash"); 18- al.add(10); 19- al.add(10.5); 20- al.add(true); 21- m1(al); 22- System.out.println(al); 23- al.add(10); 24- } 25- }</pre>	<p>ERROR!</p> <p>javac /tmp/ij10uX1D9k/GenericsCommunicationDemo.java /tmp/ij10uX1D9k/GenericsCommunicationDemo.java:9: error: incompatible types: String cannot be converted to Number al.add("Vikash"); // Compile time Error ^</p> <p>Note: /tmp/ij10uX1D9k/GenericsCommunicationDemo.java uses unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details. Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output 1 error</p>

Note: Generics concept is applicable only during the compile time but not during the run-time.

Main.java	Output
<pre>1- import java.util.*; 2- class GenericsCommunicationDemo { 3- public static void main(String[] args) 4- { 5- ArrayList al = new ArrayList<String>(); 6- al.add("Naina"); 7- al.add("Vikash"); 8- al.add("Akash"); 9- al.add(10); 10- al.add(10.5); 11- al.add(true); 12- System.out.println(al); 13- } 14- }</pre>	<p>java -cp /tmp/ij10uX1D9k GenericsCommunicationDemo [Naina, Vikash, Akash, 10, 10.5, true]</p>

Here Reference is taken care by Compiler at compile time. So Compiler will see and consider Non-generic version of ArrayList i.e., only references and we can add any object to it. Objects creation will be taken care at the run-time by JVM and in Generics after compilation Generics are removed. So No run-time exception is observed.

Main.java	Output
<pre>1- import java.util.*; 2- class GenericsCommunicationDemo { 3- public void m1(ArrayList<String> l) 4- { 5- } 6- } 7- public void m1(ArrayList<Integer> l) 8- { 9- } 10- } 11- public static void main(String args[]){ 12- } 13- } 14- } 15- }</pre>	<p>ERROR!</p> <p>javac /tmp/oz6LrqB6rA/GenericsCommunicationDemo.java /tmp/oz6LrqB6rA/GenericsCommunicationDemo.java:7: error: name clash: m1(ArrayList<Integer>) and m1(ArrayList<String>) have the same erasure public void m1(ArrayList<Integer> l) ^</p> <p>1 error</p>