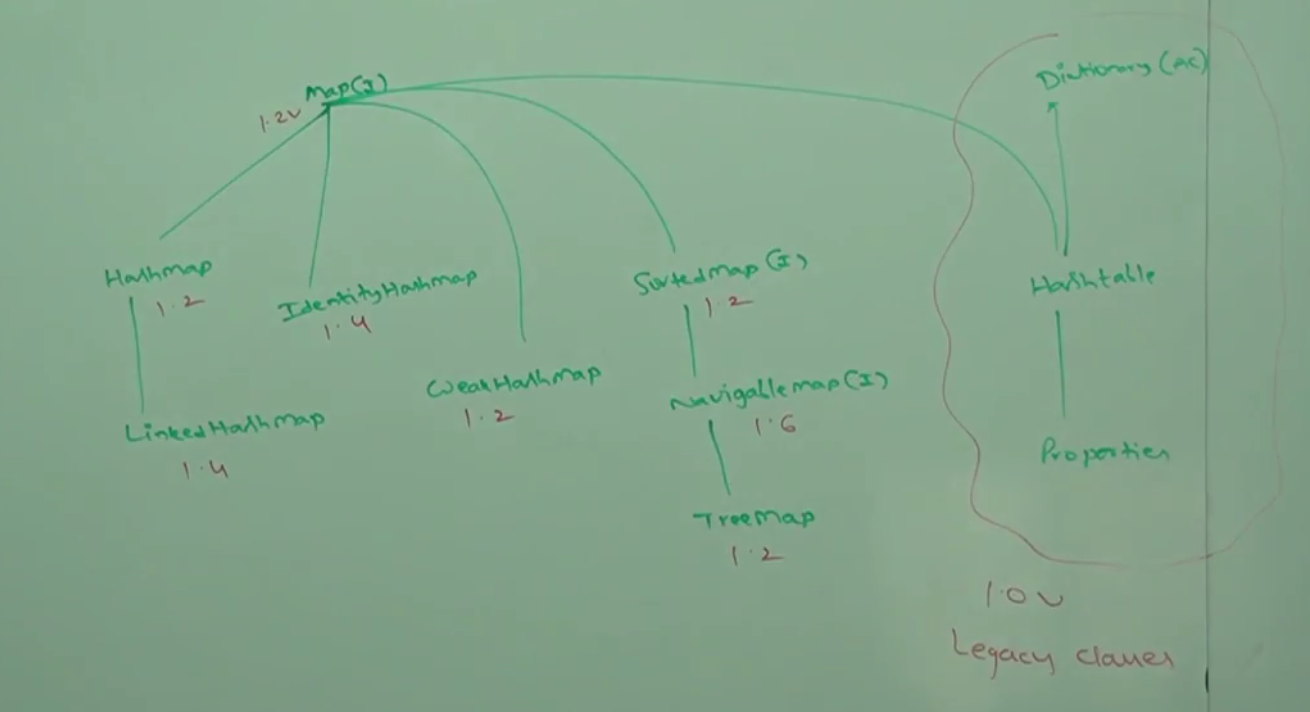
**Session-11**

**Map-HashMap**

|  |  |
| --- | --- |
| Map(1.2)(I) | 1. 🡸HashMap(1.2) 2. 🡸IdentityHashMap(1.4) 3. 🡸SortetMap(1.2) 4. 🡸HashTable(1.0) |
| Map(1.2) | 🡸HashMap | 🡸LinkedHashMap(1.4) |
| Map(1.2) | 🡸IdentityHashMap | 🡸WeakHashMap(1.2) |
| Map(1.2) | 🡸SortetMap | 🡸NavigableMap(1.6) | 🡸TreeMap(1.2) |
| Map(1.2) | 🡸Dictionary(AC) 🡸 HashTable | 🡸Properties(1.0) |



* Map is **not** a child Interface of collection because collection talks about a group of individual object as a single entity and there is concept of key-value pair in any of the Collection object, where as Map talks about a group of object as a single entity in form of Key-value pair.
* If we want to represent a group of object as Key-Value pair then we should go for Map.(StudentRollNo- StudentName),(MoNum- Name),(EmpId-EmpName)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Key | Value | | 101 | Durga | | **102** | **Arun** | | 103 | Tarun | | **104** | **Arun** | | 105 | Shilpi | | * In the Map Key and value both are object only * Duplicate keys are not allowed but values can be duplicate * Each key-value pair is called **entry** of the Map, so in other words we can say that Map is nothing but a collection of entries objects. |

Note: Map (I) concept is nowhere related to Collection and it has their own method. For example if want to add any object in the collection then we call add (object obj) method in the collection and it can have only one object and this method cannot be used in case of Map because it has two object to be added as an entries that is why Map contains its own methods .

**Map Interface Methods:**

|  |  |
| --- | --- |
| **Object** put(Object key, Object value)  m.put(101,Durga);  m.put(102,Shiva);  m.put(101,Ravi); | To add one key-value pair to the Map, Here the put method return type is object which means if the key is already present then old value will be replaced with new value and returns old value.  Here one entry is already there [101,Durga] and now we are trying to add duplicate key so in this case the value against this key will be changed and now the new entry will be [101,Ravi]. |
| **void putAll(**Map m**)** | To add a group of key-value pair i.e. add one map to another map |
| **Object get(**Object key**)** | Returns the value associated with specified key |
| **Object remove(**Object key**)** | Removes the entry associated with the specified key |
| **boolean contains(**Object ke**y)** | Returns true if the key available |
| **boolean contains(**Object value**)** | Returns true if the value available |
| **boolean** isEmpty**()** | Checks whether map is having any entries or not |
| **int** size**()** | Returns the size of Map |
| **Void** clear**()** | All key value pair will be removed |

|  |
| --- |
| **Set keyset ():** If we want to get the total number of key set from the Map entries then keyset () will be used and it will return Set. Since Map does not accept any duplicate keys and all the keys available in the Map entries are unique that is why it returns the Set because in Set also all the values are unique and duplicates are not allowed. So these methods returns only set of keys. |
| **Collection values ():** If we want get only values from the map then we can use values () method and since values can be duplicate that is why its returns type is also generic (Collection) |
| **Set entrySet ():** If we want to get the total number of the Map entries then **entrySet** () will be used and it will return a Set. Since Map does not accept any duplicate keys and all the keys available in the Map entries are unique that is why it returns the Set because in Set all the values are unique and duplicates are not allowed. So these methods returns only set of entries. And we have to call this method on Map object. |

**Collection view of Map**: As we can see the above three methods are nothing but collection only, the only difference is that these three methods are applied on Map object that is why these three methods are called **Collection view of Map i.e. these methods provided view of Map in the form of collection.**

**Entry (I) Interface:**

As we know that a Map is nothing but a group of key-value pair and each key-value pair is nothing but an entry of the map and hence Map is nothing but a collection or group of entry objects. Without existing Map object, there is no chance of exisiting Entry object. Hence Entery interface is defined inside Map interface.

**Q:** Entry is the part of Map or not? If Map is not there then is there any chance of entry.

**A:** Yes Entry is the part of Map and if Map is not there then there is no chance of entry also, that is why Entry (I) interface is an inner interface of Map interface.

|  |  |
| --- | --- |
| Interface Map{  ..  Interface Entry{  Object getKey();  Object value();  Object setValue(Object newValue);  }  } | So the Entry is an inner interface of the map which contains three methods. Since an entry is nothing but key-value pair that is why we have only three methods can be applied only Entry objects. |

Q: What is the first implementation of Map? Ans: HashMap

**HashMap (class):**

1. This is the first implementation class of Map interface
2. The HashMap concept is implemented based on HashTable so the underlying data structure of HashMap is HashTable.
3. The key-value pair is inserted on the base of hashCode of keys so the insertion order is not preserved.
4. Duplicate key are not allowed but duplicate values can be allowed
5. Heterogeneous objects for both keys and values are allowed
6. **null** **key** is allowed but only once
7. **null** **values** is allowed for any number of times
8. **HashMap** by default implements Serializable & Clonable interface but not RandomAceess
9. **HashMap** is the best choice of our frequent operation is search operation

**Constructors: Data structure of HashMap**

|  |  |
| --- | --- |
| HashMap m = new HashMap(); | Default initialCapacity = 16 & Default Fill Ratio = 0.75 |
| HashMap m = new HashMap(int initialCapacity); | Creates the HashMap of give capacity |
| HashMap m = new HashMap(int initialCapacity Float fillRatio); | Creates the HashMap of give capacity & Fill Ratio |
| HashMap m = new HashMap(Map m); | Converts the Map m into HashMap |

|  |  |
| --- | --- |
| **package** com.map.hasmap.concept;  **import** java.util.Collection;  **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.Map;  **import** java.util.Set;  **public** **class** HashMapConcept {  **public** **static** **void** main(String[] args) {  HashMap<String, Integer> m = **new** HashMap<String, Integer>();  m.put("Chiranjivi", 700);  m.put("Bala", 800);  m.put("Vanky", 900);  m.put("Naga", 200);  System.***out***.println(m);// {K=V, K=V, K=V...}  System.***out***.println(m.put("Chiranjivi", 1000));// Print old value-700  Set<String> keySet = m.keySet();  System.***out***.println(keySet);// Map Key Set [Key, key, Key, Key]  Collection<Integer> c = m.values();  System.***out***.println(c); // Map Value collection: [value, value, value, value]  Set entrySet = m.entrySet();  System.***out***.println(entrySet);// [K=V, K=V, K=V, K=V]  Iterator itr = entrySet.iterator();  **while** (itr.hasNext()) {  Map.Entry entry = (Map.Entry) itr.next();  System.***out***.println(entry.getKey() + "=" + entry.getValue());    **if**(entry.getKey().equals("Chiranjivi")) {  entry.setValue(1500); /**/ Setting new values for key "Chiranjivi"**  }  }  System.***out***.println(m);  }  } | Originam Map- {Chiranjivi=700, Vanky=900, Naga=200, Bala=800}  putting duplicate Key700  Only keyset of Map-[Chiranjivi, Vanky, Naga, Bala]  Only Values of Map-[1000, 900, 200, 800]  EntrySet of Map[Chiranjivi=1000, Vanky=900, Naga=200, Bala=800]  **Iterating HashMap**  Chiranjivi=1000  Iterating HashMap  Vanky=900  Iterating HashMap  Naga=200  Iterating HashMap  Bala=800  Final Map-{Chiranjivi=1500, Vanky=900, Naga=200, Bala=800} |

**Differences between HashMap & HashTable**

|  |  |
| --- | --- |
| **HashMap** | **HashTable** |
| every method present inside HashMap is non synchronized | Synchronized |
| Multiple thread are allowed to operate that is why not thread safe | Thread safe |
| Thread are required to wait get lock on HashMap object that is why performance is high | Thread has to wait to get lock on HashTable object to complete its execution that is why performance is low |
| Null key (only once) & null values (multiple times) is allowed | Null key and Null values are not allowed at all otherwise we will get NullPointerException |
| Introduced in 1.2V | 1.0V |

**How to get Synchronized version of HashMap object:**

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

|  |
| --- |
| HashMap m = new HashMap(); // Non Synchronized  Map m1 = Collections.synchronizedMap(m); // Synchronized |

**LinkedHashMap(c):** Now let’s discuss about LinkedHashMap which exactly same like HashMap with a very small difference

**It is the child class of HashMap**.

It is exactly same as HashMap (including methods and constructors), except the following differences.

|  |  |
| --- | --- |
| **HashMap** | **LinkedHashMap** |
| Based on HashTable | Based on LinkedList & HashTable i.e. the combination of (LL+HS) i.e. it is hybrid data structure. |
| Insertion order is not preserved and it is based on hashCode of keys | Insertion order is Preserved |
| Introduced in 1.2V | 1.4V |

In the above HashMap program if we replace HashMap with LinkedHashMap then we will get output exactly same as we insert the record.

|  |
| --- |
| **package** com.map.hasmap.concept;  **import** java.util.Collection;  **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.LinkedHashMap;  **import** java.util.Map;  **import** java.util.Set;  **public** **class** HashMapConcept {  **public** **static** **void** main(String[] args) {  LinkedHashMap<String, Integer> m = **new** LinkedHashMap<String, Integer>();  m.put("Chiranjivi", 700);  m.put("Bala", 800);  m.put("Vanky", 900);  m.put("Naga", 200);  System.***out***.println("Originam Map- " + m);// {K=V, K=V, K=V...}  System.***out***.println("putting duplicate Key" + m.put("Chiranjivi", 1000));// Print old value-700  Set<String> keySet = m.keySet();  System.***out***.println("Only keyset of Map-" + keySet);// Map Key Set [Key, key, Key, Key]  Collection<Integer> c = m.values();  System.***out***.println("Only Values of Map-" + c); // Map Value collection: [value, value, value, value]  Set entrySet = m.entrySet();  System.***out***.println("EntrySet of Map" + entrySet);// [K=V, K=V, K=V, K=V]  Iterator itr = entrySet.iterator();  **while** (itr.hasNext()) {  Map.Entry entry = (Map.Entry) itr.next();  System.***out***.println("Iterating HashMap");  System.***out***.println(entry.getKey() + "=" + entry.getValue());  **if** (entry.getKey().equals("Chiranjivi")) {  entry.setValue(1500);// Setting new values for key "Chiranjivi"  }  }  System.***out***.println("Final Map-" + m);  }  } |

Output:

|  |  |
| --- | --- |
| -Originam Map- {Chiranjivi=700, Bala=800, Vanky=900, Naga=200}  -putting duplicate Key700  -Only keyset of Map-[Chiranjivi, Bala, Vanky, Naga]  -Only Values of Map-[1000, 800, 900, 200]  -EntrySet of Map[Chiranjivi=1000, Bala=800, Vanky=900, Naga=200]  -Iterating HashMap  Chiranjivi=1000 | Iterating HashMap  Bala=800  Iterating HashMap  Vanky=900  Iterating HashMap  Naga=200  -Final Map-{Chiranjivi=1500, Bala=800, Vanky=900, Naga=200} |

**Note:** LinkedHashSet and LinkedHashMap are commonly used for developing cache based applications.

**Difference between (==) operator and .equals () method:**

In general **(==)** operator meant of reference (address) comparison, where **.equals ()** meant of content comparison.

Example:

|  |  |
| --- | --- |
| Integer a1= new Integer(10);  Integer a2= new Integer(10);  System.out.println(a1==a2); // False  System.out.println(a1.equals(a2));// true | Here two integer object a1 and a2 got created with the same content but both the object has created at different object because we have used new operator.  So when we compare a1 & a2 using (==) operator then it return false while when we compare a1 and a2 using .equals() method then it return true. |

The above concept will be used in **IdentiHashMap** implementation class of HashMap Interface.

**IdentiHashMap (class):**It is exactly same as HashMap, with small difference.

|  |  |
| --- | --- |
| **package** com.map.Identityhasmap.concept;  **import** java.util.HashMap;  **public** **class** IdentityHashMapConcept {  **public** **static** **void** main(String[] args) {  HashMap m = **new** HashMap();  Integer a1 = **new** Integer(10);  Integer a2 = **new** Integer(10);  m.put(a1, "Arun");  m.put(a2, "Tarun");  System.***out***.println(m);  }  }  **Output:**  {10=Tarun} | **package** com.map.Identityhasmap.concept;  **import** java.util.HashMap;  **import** java.util.IdentityHashMap;  **public** **class** IdentityHashMapConcept {  **public** **static** **void** main(String[] args) {  IdentityHashMap m = **new** IdentityHashMap();  Integer a1 = **new** Integer(10);  Integer a2 = **new** Integer(10);  m.put(a1, "Arun");  m.put(a2, "Tarun");  System.***out***.println(m);  }  }  **Output:**  {10=Arun, 10=Tarun} |
| **Explanation:**  Here We have created two different Integer object (a1 & a2) with the same content.  These objects have been used as key in HashMap. As per the above program it looks like both a1, a2 has different reference or address so while using in HashMap as key then we should get two different entries of HashMap. But when we print then we found that the first entry value got replaced with second entry value:  Reason: Here in the normal HashMap JVM uses **.equals () method** which compare content of both the object and hence when we put another entry in the HashMap then due to same content it seems as duplicate key and hence it replaces the value old value with the another duplicate key value and that is why we have got one entry only. | **Reason:**  Here in case of IdentityHashMap JVM uses **(==) operator** to compare duplicate key entries and that is why both the keys have different references or address.  Hence while making two entries with the same content because having different references or address it seems to be two different key and that is why we have got two entries. |

So IdentiHashMap exactly same as (including methods & constructors) accept the following difference.

In the case of Normal HashMap JVM will use **a1.equals (a2)** method to identify duplicate keys, which meant for content comparison. A1 & a2 are duplicate keys because **a1.equals (a2) returns true**

But in case of IdentiHashMap JVM will **use (==) operator** to identify duplicate keys which is meant for reference or address comparison.

If replace HashMap with IdentityHashMap then a1 & a2 are not duplicate keys because a1==a2 returns false. In this case output will have two entries.

**WeakHashMap (Class):**

Let’s say we have one object which does not have any reference then in case this object is eligible for garbage collection. Now the Garbage collector mark this object for destroy. But before destroying this object the garbage collector call **finalize ()** method so that if any activity associated to the object like db connection and other clean up activity can be performed by the garbage collector in the finalize() method and once finalize() complete its execution then immediately GC destroy the object.

So in-short GC called finalize () method just before destroying the unreferenced object to perform cleanup activity.

|  |  |
| --- | --- |
| **package** com.map.weekhashmap.concept;  **public** **class** Temp {  @Override  **public** String toString() {  **return** "temp";  }  @Override  **protected** **void** finalize() **throws** Throwable {    System.***out***.println("Finalize method called..");  }  } | **package** com.map.weekhashmap.concept;  **import** java.util.HashMap;  **public** **class** WeekHashMapConcept {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  HashMap<Temp, String> m = **new** HashMap<Temp, String>();  Temp t = **new** Temp();  m.put(t, "Durga");  System.***out***.println("Before Making t-null" + m);  t = **null**;  **System.*gc*();**  Thread.*sleep*(5000);  System.***out***.println("After making t-null" + m);  }  }  Output:  Before Making t-null{temp=Durga}  After making t-null{temp=Durga} |

**Now let’s use WeakHashMap instead of HashMap**

|  |  |
| --- | --- |
| **import** java.util.HashMap;  **import** java.util.WeakHashMap;  **public** **class** WeekHashMapConcept {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  // HashMap<Temp, String> m = new HashMap<Temp, String>();  WeakHashMap<Temp, String> m = **new** WeakHashMap<Temp, String>();  Temp t = **new** Temp();  m.put(t, "Durga");  System.***out***.println("Before Making t-null" + m);  t = **null**;  System.*gc*();  Thread.*sleep*(5000);  System.***out***.println("After making t-null" + m);  }  }  output  Before Making t-null{temp=Durga}  Finalize method called...  After making t-null{} | Here in case of HashMap we have created one entry where Temp Class object (t) is put as key and value as “Durga” i.e. now t is associated with HashMap value “Durga” and Printed m.  Now after associating t with HashMap object, we have made Temp class object t as null i.e. now t object becomes unreferenced object and hence according to rule it is eligible for GC. But when GC comes to destroy this Temp object (t) then it observes that even though t has no reference now but it is associated with HashMap and due to this HashMap does not allow the GC to destroy Temp object i.e. HashMap dominates on GC  But in case of WeakHashMap, it could not dominate on GC and then in that case GC destroy the unreferenced Temp class object (t) by calling finalize() method. Hence we have got the output as given. |

So WeakHashMap is exactly same Normal HashMap with some difference.

In the HashMap even though object does not have any reference, it is not eligible for GC it is associated with HashMap i.e. HashMap dominates GC.

But in the case of WeakHashMap, if object does not contain any references it is eligible for GC even though object is associated with WeakHashMap i.e. GC dominates WeakHashMap.

In the above example temp object is not eligible for GC because associated HashMap and in this case in output both time we will get HashMap entry.

In the above program if replace HashMap with WeakHashMap then temp object is eligible for GC and in this case output is as given in the example.

So HashMap dominates GC but WeakHashMap does not dominated GC that is why it is called WeakHashMap.

**SortedMap (I)**

It is the child interface of Map.

If we want to represent a group of key-value pair according to some sorting order of keys then we should go for sorted map.

Here sorting is based on the key but not based on value.

Note: it is very similar to Set (I) 🡸SortedSet (I) where a group of objects are inserted according to some sorting order and since it is set so duplicate values are not allowed.

In the same way we have SortedMap which has the same concept and methods just like SortedSet but here instead of value entries of shorted map inserted based on sorting order of key.

SortedMap defines the following specific methods.

|  |  |
| --- | --- |
| Object firstKey(); | Return the first key of sortedMap |
| Object lastKey(); | Return the last key of sortedMap |
| SortedMap headMap(Object key) | Return the total map entries which key is less than passed key inside headMap(key) |
| SortedMap tailMap(Object key) | Return the total map entries which key is greater than equal to the passed key inside tailMap(key) |
| SortedMap subMap(Object key1, Object key2) | Return the total map entries less than equal to key1 and greater than key2 |
| Comparator comparator() | Return null if map is sorted |
| |  |  | | --- | --- | | 101 | A | | 103 | B | | 104 | C | | 107 | D | | 125 | E | | 136 | F | | firstKey() 🡺 101  lastKey() 🡺 136  headMap(107) 🡺{101=A, 103=B, 104=C}  tailMap(107) 🡺{107=D, 125=E, 136=F}  subMap(103,125) 🡺 {103=B, 104=C, 107=D}  comparator()🡺 null if map is sorted |

**TreeMap (class)**

1. It is the implementation of NavigableInterface (I)
2. It is implemented based on Red-Black Tree i.e. the underlying data structure of TreeMap is Red-Black Tree.
3. Element will inserted according to some sorting order of keys and Insertion order is not preserved
4. Duplicates are not allowed but value can be duplicated
5. If we are depending on default natural sorting order then keys should be homogeneous & comparable otherwise we will get run time exception saying class-cast-exception
6. If we are defining our own sorting by using comparator then keys need not be homogeneous and comparable, we can take heterogeneous non-comparable objects also
7. Whether we are depending on default natural sorting order or customized sorting order, there are no restrictions for values; we can take heterogeneous, non-comparable object values also.

**Null Acceptance in TreeMap:**

**Condition-1:** for non empty TreeMap if we are trying insert an entry with null key then we will get Runtime Exception saying NullPointerException because while trying to insert null key then this key will be compared with the previous available key and then NPE.

**Condition-2:** For empty TreeMap as the first entry with null key is allowed but after inserting that entry if we are trying to insert any other entry then we will get RTE saying NPE.

**The above null acceptance rule applicable until 1.6 V only from 1.7V null is not allowed for key. But for values we can use null any number of times, there is no restriction whether it is 1.6 V or 1.7 V.**

**Constructors – TreeMap**

|  |  |
| --- | --- |
| TreeMap t = new TreeMap() | For default natural sorting order |
| TreeMap t = new TreeMap(Comparator c) | For customized sorting order |
| TreeMap t = new TreeMap(Map m) | Created a TreeMap on given Map object |
| TreeMap t = new TreeMap(SortetMap m) | Created a TreeMap on given SortedMap object |

Example:

|  |  |
| --- | --- |
| **package** com.map.treemap.concept;  **import** java.util.TreeMap;  **public** **class** TreeMapConcept {  **public** **static** **void** main(String[] args) {  TreeMap<Object, Object> tm = **new** TreeMap<Object, Object>();  tm.put(100, "XXX");  tm.put(103, "ZZZ");  tm.put(101, "YYY");  // tm.put("AAA", "**XXX**"); //java.lang.ClassCastException:  // tm.put(null, "YYY"); //java.lang.NullPointerException  System.***out***.println(tm);  }  }  Output: {100=XXX, 101=YYY, 103=ZZZ} | Example to compare Key integer |
| **package** com.map.treemap.concept;  **import** java.util.Comparator;  **public** **class** MyComparator **implements** Comparator<Object> {  @Override  **public** **int** compare(Object obj1, Object obj2) {  // **TODO** Auto-generated method stub  String s1 = obj1.toString();  String s2 = obj2.toString();  **return** s2.compareTo(s1);  }  } | **package** com.map.treemap.concept;  **import** java.util.TreeMap;  **public** **class** TreeMapWithComparator {  **public** **static** **void** main(String[] args) {  MyComparator myComparator = **new** MyComparator();  **TreeMap<String, Integer> tm = new TreeMap<String, Integer>(myComparator);**  tm.put("YYY", 10);  tm.put("AAA", 10);  tm.put("DDD", 10);  tm.put("ZZZ", 10);  System.***out***.println(tm);  }  }  **Output :**  {ZZZ=10, YYY=10, DDD=10, AAA=10} |

So we have covered [HashMap, LinkedHashMap, IdentityHashMap, and WeakHashMap & TreeMap]

Now let’s discuss two legacy classes in the Map (HashTable and Property)

**HashTable: (Class)**

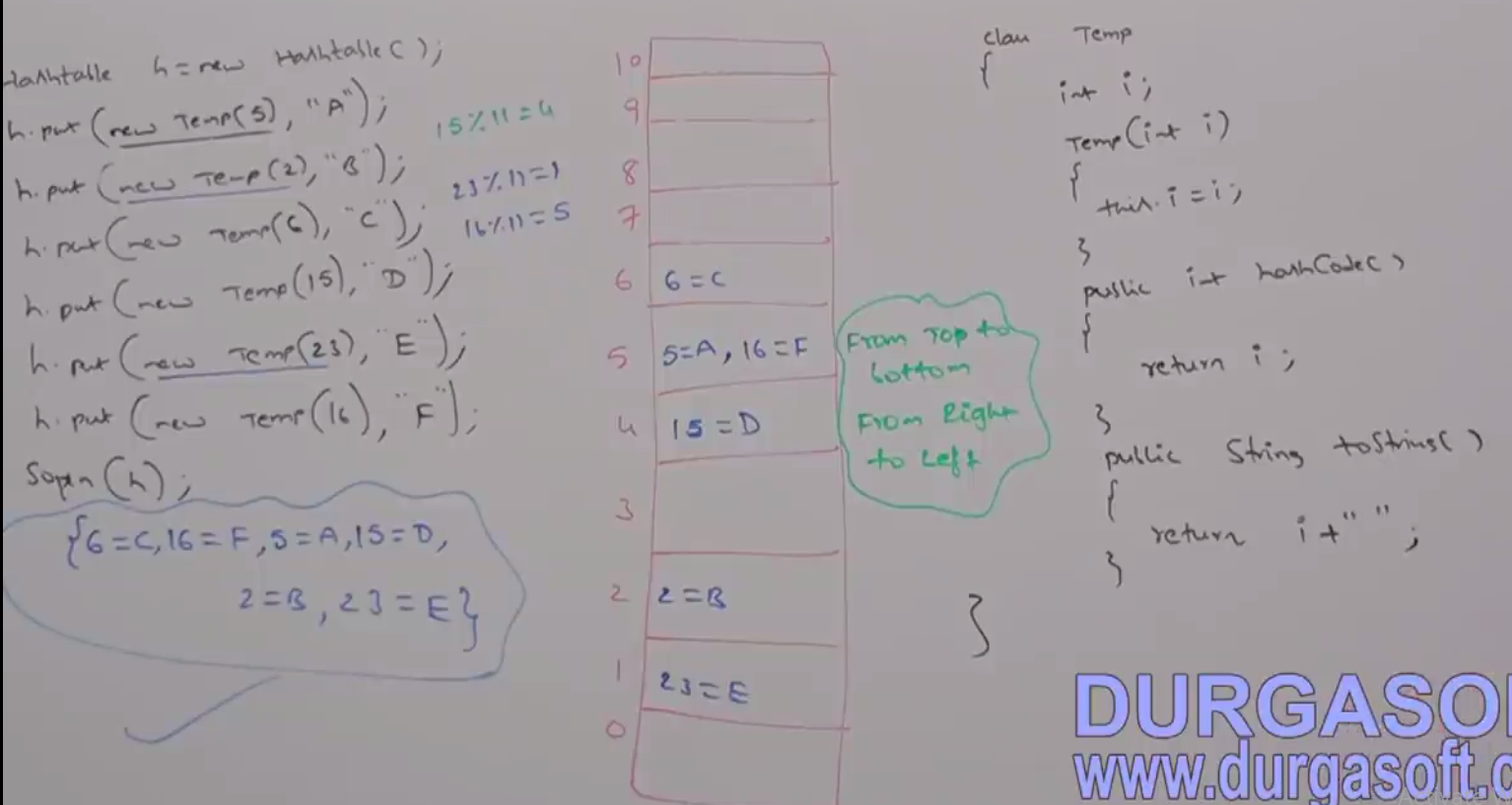
1. The underlying data structure for HashTable is HashTable
2. Insertion order is not preserved and it is based on hashCode of keys
3. Duplicate keys are not allowed and value can be duplicated
4. Heterogeneous objects are allowed for both keys and values
5. Null is not allowed for both key and value otherwise we will get runtime exception saying NPE
6. It implements Serializable & Clonable Interfaces but not RandomAceess Interface
7. Every methods present in HashTable is synchronized and hence HashTable object thread-safe
8. HashTable is the best choice is frequent operation is search operation

**Constructors:**

|  |  |
| --- | --- |
| HashTable m = new HashTable (); | Default initialCapacity = 11 & Default Fill Ratio = 0.75 |
| HashTable m = new HashTable (int initialCapacity); | Creates the HashMap of give capacity |
| HashTable m = HashTable HashMap(int initialCapacity Float fillRatio); | Creates the HashMap of give capacity & Fill Ratio |
| HashTable m = new HashTable (Map m); | Converts the Map m into HashMap |

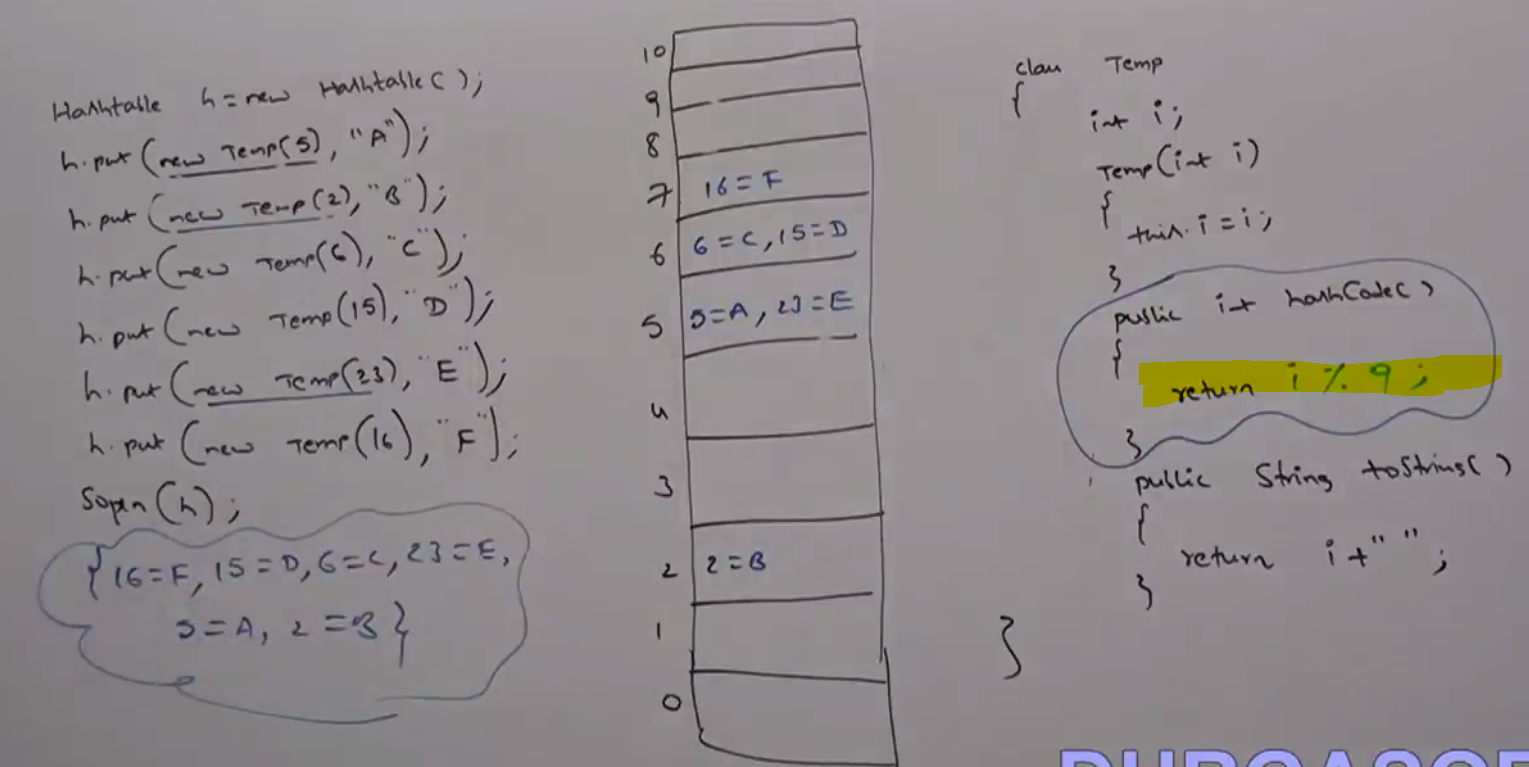
Example:

|  |  |
| --- | --- |
| **package** com.map.hastable.concept;  **import** java.util.Hashtable;  **public** **class** HashTableConcept {  **public** **static** **void** main(String[] args) {  Hashtable ht = **new** Hashtable();  ht.put(5, "A");  ht.put(2, "B");  ht.put(6, "C");  ht.put(15, "D");  ht.put(23, "E");  ht.put(16, "F");  System.***out***.println(ht);  }  } | Output:  {6=C, 16=F, 5=A, 15=D, 2=B, 23=E}  Here the hashCode is calculated by JVM only and based on the hashCode the entries are going into buckets.  **Temp.java object will used as key in Hashtable**  **package** com.map.hastable.concept;  **public** **class** Temp {  **int** i;  **public** Temp(**int** i) {  **super**();  **this**.i = i;  }  @Override  **public** String toString() {  **return** i + "";  }  @Override  **public** **int** hashCode() {  **return** i;  }  }  Here we have created Temp class and **override hashCode () method and returning our own hashCode (i).**  So here we are creating our own custom hashCode and then according to this hashCode only the entries are going into buckets |
| **package** com.map.hastable.concept;  **import** java.util.Hashtable;  **public** **class** HashtableCustomHashCode {  **public** **static** **void** main(String[] args) {    Hashtable ht = **new** Hashtable();  ht.put(**new** Temp(5), "A");  ht.put(**new** Temp(2), "B");  ht.put(**new** Temp(6), "C");  ht.put(**new** Temp(15), "D");  ht.put(**new** Temp(23), "E");  ht.put(**new** Temp(16), "F");    System.***out***.println(ht);  }  }  Output:  {6=C, 16=F, 5=A, 15=D, 2=B, 23=E} |
| So as per the program we are putting different object of Temp class using[ new Temp(5)] in the Hashtable put method which calls the constructor of Temp class and assign value 5 into int i which in-turn getting returned by hashCode() method. The entries will be inserted into Hashtable buckets based on this returned hashCode value which is shown in the below image. Now once all the entries gets inserted into buckets then the display of the entries will be ;  -From Top to Bottom  -From Right to Left | |



Note: Now instead of calculating hashCode as (i) let’s calculate the hashCode (i % 9). In this case the output will be changed.

|  |  |
| --- | --- |
| **package** com.map.hastable.concept;  **import** java.util.Hashtable;  **public** **class** HashtableCustomHashCode {  **public** **static** **void** main(String[] args) {  Hashtable ht = **new** Hashtable();  ht.put(**new** Temp(5), "A");  ht.put(**new** Temp(2), "B");  ht.put(**new** Temp(6), "C");  ht.put(**new** Temp(15), "D");  ht.put(**new** Temp(23), "E");  ht.put(**new** Temp(16), "F");    System.***out***.println(ht);  }  }  Output:  {16=F, 15=D, 6=C, 23=E, 5=A, 2=B} | **package** com.map.hastable.concept;  **public** **class** Temp {  **int** i;  **public** Temp(**int** i) {  **super**();  **this**.i = i;  }  @Override  **public** String toString() {  **return** i + "";  }  @Override  **public** **int** hashCode() {  **return** i % 9;  }  }  So as we can see that once the hashCode got changed then the output also got changed accordingly |



Note: As we know that if we don’t define capacity in the Hashtable then the initial capacity will be 11 and then (0 to 10) buckets will be created and then according to hashCode the element will be inserted into bucket and the element will be printed ( From Top 🡺 Bottom and Right 🡺Left)

Now in the above program if we just change the capacity or Hashtable then the output also will be changed.

|  |  |
| --- | --- |
| **package** com.map.hastable.concept;  **import** java.util.Hashtable;  **public** **class** HashtableCustomHashCode {  **public** **static** **void** main(String[] args) {  **Hashtable ht = new Hashtable(25);**  ht.put(**new** Temp(5), "A");  ht.put(**new** Temp(2), "B");  ht.put(**new** Temp(6), "C");  ht.put(**new** Temp(15), "D");  ht.put(**new** Temp(23), "E");  ht.put(**new** Temp(16), "F");  System.***out***.println(ht);  }  }  The previous out with default capacity: is  {6=C, 16=F, 5=A, 15=D, 2=B, 23=E} | **package** com.map.hastable.concept;  **public** **class** Temp {  **int** i;  **public** Temp(**int** i) {  **super**();  **this**.i = i;  }  @Override  **public** String toString() {  **return** i + "";  }  @Override  **public** **int** hashCode() {  **return** i;  }  }  The out output with initialCapacity 25:  {23=E, 16=F, 15=D, 6=C, 5=A, 2=B} |

So if capacity gets changed then the output also will be changed.

**Properties Class:**

In our day to today programming anything is getting change frequently then it is not recommended to hard code in Java. For example if at every three months if we have to change Password of data base as security concern and if we have written the line of code in java then whenever we change the password in the java file then we have to do following activity for reflecting the changes in the server side.

1. Compile the code
2. Rebuild (in ear or war)
3. Redeploy the war in the server
4. If changes does not reflect then restart the server also

So for a small change we have to do lot of activities and minimum 2 to 3 hours will take and during that time application will be down and will impact on business also.

So the solution of the above problem is we will create one properties file and variable is getting change frequently like (mob number, password) will be configured in the properties file only. Now read these values in the java program and whenever there is any change then just change in the property file and re-deploy the application in the server to reflect the change that’s it. So properties file plays very important role in the real time application.

Q: So far we have discussed about properties file but where the Properties object role comes in the application?

A: When we read the properties file in the java file then to hold the data of these properties file Properties class object is required i.e. Properties class java object holds these file values. So first of all will load this properties file in the Properties class object then according to requirement will read those values from the object.

In our program if anything which changes frequently (like username, password, mail id, mob num etc) are not recommended to hard code in java program because if there is any change then to reflect that change, recompilation, rebuild and re-deploy application are required, even sometimes server re-start is also required, which creates a big business impact to the client. We can overcome this problem by using properties file. Such type of variable things we have to configure in the properties file. From that properties file, we have to read into java program and we can use those properties.

The main advantage of this approach is if there is change in properties file, then to reflect that change just re-deployment is enough, which won’t create any business impact to the client.

We can use java properties object to hold properties which are coming from properties file.

**Note:** In normal map (HashMap, Hashtable, TreeMap) key and value can be any type but in the case of properties key and value should be String type only.

Constructor: Only one

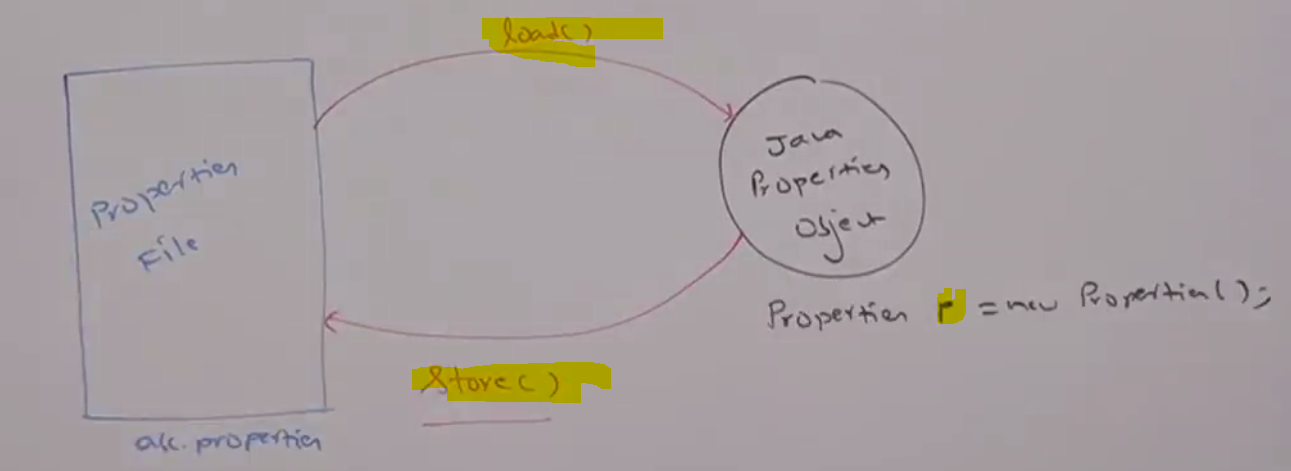
**Properties prop = new Properties ()**

**Methods: Three m**

|  |  |
| --- | --- |
| String **getProperties**(String **pname**) | Returns the value associated with property pname (key); if the specified property is not available then it returns null. it is similar to **get(Object key)** of HashMap and other Map classes |
| String **setProperties**(String **pname**, String **pvalue**) | Set a new property associated and if the specified property (pname) is already available then old value will be replaced with new value and returns old value and it is very similar to **setValue (Object value)** of HashMap and Entry interface. |
| Enumeration **propertyName**() | Returns the entire property name (key) present in properties object. So once we get all property associated in the file then by using **getPropertie(pname)** we can retrieve the value |

**Apart from these three methods we have two more method:**

|  |  |
| --- | --- |
| **void load(**InputStream **is)** | To load all properties from properties file into java properties object |
| **Void store(**OutputStream **os**, String **comment)** | To store all properties from the java properties object into properties file( reverse of load() method) |

****

**Note:** Generally to mention a file as properties file usually everyone put (filename. properties) just for readable purpose. This extenuation could have any name instead of (.properties) like abc.txt, abc.xml, abc.arun etc. The java follows UNIX OS convention for any file in java IO where extenuation is not important.

**Example**

As in the above program created one abc.properties file .

In the java program created one object of Properties class. Using the FileInputStream converted the abc.properties file into byte streams.

Now load this byte stream into Properties object (prop) using load () method.

|  |  |
| --- | --- |
| abc.properties  xyz=789  abc=123  def=456 | **package** com.map.properties.concept;  **import** java.io.FileInputStream;  **import** java.util.Properties;  **public** **class** PropertiesConcept {  **public** **static** **void** main(String[] args) **throws** Exception {  Properties prop = **new** Properties();  FileInputStream fis = **new** FileInputStream("abc.properties");  prop.load(fis);  System.***out***.println(prop);// {xyz=789, abc=123, def=456}  String s = prop.getProperty("xyz");  System.***out***.println("xyz value:" + s);  // prop.setProperty("xyz", "000"); //  // prop.setProperty("bbb", "999");  // System.out.println(prop);  // FileOutputStream fos = new FileOutputStream("abc.properties");  // prop.store(fos, "Updted by Arun");  }  } |

Now when we try to print (porp) object then it will print the properties in the map form.

So in the above program we have loaded abc.properties file into Properties java object (prop)

Now in the remaining part of the program we will set new properties from java to abc.properties file and then will load Properties java object into abc.properties file.

Let’s uncomment the commented code in the above program a

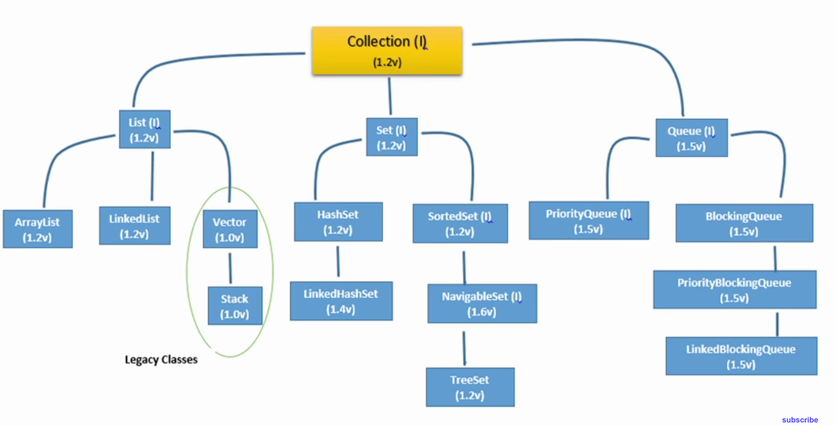
|  |  |
| --- | --- |
| **package** com.map.properties.concept;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.util.Properties;  **public** **class** PropertiesConcept {  **public** **static** **void** main(String[] args) **throws** Exception {  Properties prop = **new** Properties();  FileInputStream fis = **new** FileInputStream("abc.properties");  prop.load(fis);  System.***out***.println(prop);// {xyz=789, abc=123, def=456}  String s = prop.getProperty("xyz");  System.***out***.println("xyz value:" + s);  // -loading java object into abc properties file  prop.setProperty("xyz", "000"); // overwrite xyz value with 000  prop.setProperty("bbb", "999");// create a new properties in the file  FileOutputStream fos = **new** FileOutputStream("abc.properties");  // converting abc.properties file conent into output stream object  prop.store(fos, "Updted by Arun");  // loading the object into file using store() method  System.***out***.println(prop); // printing the updated file again  }  } | Before updating file:  xyz=000  abc=123  def=456  When we run the first time the output will be  {xyz=789, abc=123, def=456}  xyz value:789  {bbb=999, xyz=000, abc=123, def=456}  Second time when we run the program again then the output and updated fill will look like as below.  {xyz=000, bbb=999, abc=123, def=456}  xyz value:000  {xyz=000, bbb=999, abc=123, def=456}  Updated abc.properties file:  #Updted by Arun  #Thu Apr 01 23:07:38 IST 2021  xyz=000  bbb=999  abc=123  def=456 |

**Program to connect database using properties file:**

|  |  |
| --- | --- |
| **package** com.map.properties.concept;  **import** java.io.FileInputStream;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.util.Properties;  **public** **class** DBConnectionUsingProperties {  **public** **static** **void** main(String[] args) **throws** Exception {  Properties prop = **new** Properties();  FileInputStream fis = **new** FileInputStream("db.properties");  prop.load(fis);  String url = prop.getProperty("url");  String user = prop.getProperty("user");  String pwd = prop.getProperty("pwd");  System.***out***.println(prop);  Connection con = DriverManager.*getConnection*(url, user, pwd);  }  } | url=https://mysql.com  user=arun  pwd=123@abc  output:  {user=arun, url=https://mysql.com , pwd=123@abc}  Exception in thread "main" java.sql.SQLException: No suitable driver found for <https://mysql.com>  Because url name is not correct |

---------------------------Map Part completed here----------------------now will start Queue of collection-------------------------

**1.5 version enhancements (Queue Interface):**



1. It is the child interface of collection
2. If we want to represent a group of individual prior to processing then we should go for queue
3. For example: Before sending sms message / email all mobile numbers / email\_Id we have to store in some data structure, and in which order we added mobile number / email, in the same order only message / email should be delivered. For this FIFO requirement Queue is the best choice.
4. Usually Queue follows FIFO order but based on our requirement , we can implement our own propriety order also (PriorityQueue)
5. From 1.5 versions onwards LinkedList class also implement Queue interface. LinkedList based implementation of queue always follows FIFO order.

**Queue Interface specific 5- methods:**

|  |  |
| --- | --- |
| boolean offer(Object obj) | To add an object into queue |
| Object peek() | Returns head (first) element of the queue. If queue is empty then this method returns null |
| Object element() | Returns head (first) element of the queue. If queue is empty then this method instead of returning null, raises RE: NoSuchElementException |
| Object poll() | It removes and return head element of queue. If queue is empty then this method returns null |
| Object remove() | It removes and return head element of queue. If queue is empty then this method instead of returning null raises RE: NoSuchElementException |

**PriorityQueue:**

If we want to represent a group of individual objects prior to processing according to some priority then we should go for Priority Queue.

1. The priority can be either default natural sorting order or customized sorting order defined by comparator.
2. Insertion order is not preserved because it is based on some priority.
3. Duplicates objects are not allowed
4. If we are depending on DNSO, then compulsory object should be homogeneous and comparable otherwise will get RTE saying ClassCastException
5. If we are defining our own sorting by comparator then object need not be homogeneous and comparable
6. Null is not allowed even as the first element also

Constructors:

|  |  |
| --- | --- |
| PriorityQueue pq = new PriorityQueue() | Creates an empty PQ with default initial capacity 11 and all object will be inserted according to DNSO |
| PriorityQueue pq = new PriorityQueue(int initialCapacity) |  |
| PriorityQueue pq = new PriorityQueue(int initialCapacity, Comparator c) |  |
| PriorityQueue pq = new PriorityQueue(SortetSet s) |  |
| PriorityQueue pq = new PriorityQueue(Collection c) |  |

Program for DNSO of PQ

|  |  |
| --- | --- |
| **package** com.collection.pqueue.concept;  **import** java.util.PriorityQueue;  **public** **class** PriorityQueueConcept {  **public** **static** **void** main(String[] args) {  PriorityQueue pq = **new** PriorityQueue();  System.***out***.println("Empty Q");  System.***out***.println("pq.peek() on Empty Q- " + pq.peek());//Null  System.***out***.println("pq.element()on EmptyQ- " + pq.element());//Exep  }  } | So peek() will return null because queue is empty.  Element() will return RTE on empty queue. So the output is  Empty Q  pq.peek() on Empty Q- null  Exception in thread "main" java.util.NoSuchElementException |

Now let’s comment peek() and element() and insert element in PQ using offer (Object obj) method.

|  |  |
| --- | --- |
| **package** com.collection.pqueue.concept;  **import** java.util.PriorityQueue;  **public** **class** PriorityQueueConcept {  **public** **static** **void** main(String[] args) {  PriorityQueue pq = **new** PriorityQueue();  // System.out.println("Empty Q");  // System.out.println("pq.peek() on Empty Q- " + pq.peek());  //System.out.println("pq.element() on Empty Q- " + pq.element());  **for** (**int** i = 0; i <= 10; i++) {  pq.offer(i);  }  System.***out***.println("Now PQ-" + pq);  System.***out***.println("pq.peek() on Empty Q- " + pq.peek());  System.***out***.println("Now PQ-" + pq);  System.***out***.println("pq.element() on PQ- " + pq.element());  System.***out***.println("Now PQ-" + pq);  System.***out***.println("pq.poll() on PQ- " + pq.poll());  System.***out***.println("Now PQ-" + pq);  System.***out***.println("pq.remove() on PQ- " + pq.remove());  System.***out***.println("Now PQ-" + pq);  }  } | Output:  Now PQ-[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  pq.peek() on Empty Q- 0  Now PQ-[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  pq.element() on PQ- 0  Now PQ-[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  pq.poll() on PQ- 0  Now PQ-[1, 3, 2, 7, 4, 5, 6, 10, 8, 9]  pq.remove() on PQ- 1  Now PQ-[2, 3, 5, 7, 4, 9, 6, 10, 8] |

**Note**: As we can see that we are not getting PQ after applying poll () and remove () method i.e. output of PQ is not in DNSO, the reason is some operating system does not support Priority Q. It is very similar to Thread Priority like because we have seen that Thread Priority is supported by some OS. And in this case we are not getting expected output.

So some platform wont’ provides proper support for Thread Priority and Priority Queue. And in this case we have to send email to provide support for this and they will install patch for that.

**Demo program of PQ for customized Priority Queue:**

|  |  |
| --- | --- |
| **package** com.collection.pqueue.concept;  **import** java.util.Comparator;  **public** **class** MyComparator **implements** Comparator<Object>{  @Override  **public** **int** compare(Object obj1, Object obj2) {    **return** obj2.toString().compareTo(obj1.toString());  }  }  Ouput: We will get reverse order of PQ  [Z, S, N, D] | **package** com.collection.pqueue.concept;  **import** java.util.PriorityQueue;  **public** **class** PQCustomSoting {  **public** **static** **void** main(String[] args) {    MyComparator myComparator = **new** MyComparator();  PriorityQueue pq = **new** PriorityQueue(15, myComparator);  pq.offer("D");  pq.offer("Z");  pq.offer("N");  pq.offer("S");  System.***out***.println(pq);  }  } |

**BlockingQueue- Class:**

**LinkedBlockingQueue- Class:**

**1.6V Enhancement in Collection Framework:**

As the part of 1.6V the following two concepts introduced in collection framework:

1. **NavigableSet: 2- NavigableMap**

**1- NavigableSet:** It is the child Interface of SortedSet and it defines several methods for navigation purposes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Collection(I) 🡸1.2V | Set(I) 🡸1.2V | SortedSet(I) 🡸1.2V | NavigableSet(I) 🡸1.6V | TreeSet (c) 1.2V |

Let’s say we have collection of fight timings:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **00.30** | **01:45** | **02:30** | **04:20** | **05:30** | **06:45** | **07:30** | **09:20** | **10:15** | **12:25** | **13:30** | **14:25** | **15:30** | **18:20** | **20:20** | **23:25** | **23:55** |

Now let’s say if want to know the very first fight timing then we can use methods available in SortedSet – **first ()**

1-If want to know the last fight timing then we can use methods available in SortedSet – **last ()**

2-If want to know various flight before 10 ‘O’ Clock then we can use methods available in SortedSet – **headSet (10:00)**

3-If want to know various flight after 10 ‘O’ Clock then we can use methods available in SortedSet – **taildSet (10:00)**

4-From Morning 7 to 10 what are various flights then we can use methods available in SortedSet – **subset (7, 12)**

So happily we can use above method available in SortedSet if we want to know the flight status based on times.

But if we want to know before 10:00 ‘O’ clock what is the last flight or after 10:00 O clock what is the first flight? So to get answer of these questions there no any methods available in SortedSet, Set, TreeSet. So to fill this gab NavigableSet introduced in 1.6V

So to answer the question like before or after means for navigation purposes Navigable set came into picture.

NavigableSet provides several methods for Navigation purposes:

1. Now if want to know the last flight before 10:00 O clock then we have method in NavigableSet: **lower (10:00); So here lower () talks about only for before**
2. Either 10 or before 10 then i.e. including 10 or before 10 what is the last flight then : **floor(10:00); Here floor() talks about at a time or before time**
3. After 10:00 ‘O’ clock what is the first flight then NavigableSet has**: higher(10:00)**
4. What is the first flight either 10 or after 10:00 ‘O’ clock then NavigableSet contains **: ceiling(10:00)**

**Hence NavigableSet contains all the methods for Navigation support which are not available in the SortedSet or Set or TreeSet.**

|  |  |
| --- | --- |
| **floor(10:00)** | Either at 10 or before 10 what is the last flight i.e. <=10:00 **🡺: Ans :- 9:20** |
| **floor(e)** | Returns highest element which is (less than or equal to) **<= e** |
|  | |
| **Lower(10:00)** | last flight before 10:00 O clock i.e. <10 🡺 **Ans: 9:20** |
| **Lower(e)** | Returns highest element which is less than e (< e) |
| **Note: Here we are heading towards lowest and we have to select highest from all lowest that is why we have used highest element in the above example.** | |
| **ceiling(10:00)** | the first flight either 10 or after 10:00 ‘O’ clock i.e. >=10 🡺 **Ans: 10:15** |
| **ceiling(e)** | Returns the lowest element which is (greater than or equal to e) > =e |
|  | |
| **higher(10:00)** | After 10:00 ‘O’ clock what is the first flight i.e. (greater than 10) > 10 🡺 **Ans : 10:15** |
| **higher(e)** | Returns the lowest element which greater than e (> e) |
| **Note: Here we are heading towards highest element (time) and we have to select lowest from all highest element that is why we have used lowest element in the above example.** | |

|  |  |
| --- | --- |
| pollFirst() | Removes and returns first element |
| pollLast() | Removes and returns last element |
| descendingSet() | It returns NavigableSet in reverse order |

Note: These Methods can be applied on TreeSet class only because TreeSet class is the implementation class of Navigable and SortedSet

|  |
| --- |
| **package** com.collection.navigableset.concept;  **import** java.util.TreeSet;  **public** **class** NavigableTreeSetConcept {  **public** **static** **void** main(String[] args) {  TreeSet<Double> t = **new** TreeSet<Double>();  t.add(00.30);  t.add(01.45);  t.add(02.30);  t.add(04.20);  t.add(05.30);  t.add(05.45);  t.add(07.30);  t.add(09.20);  t.add(10.15);  t.add(12.25);  t.add(13.30);  t.add(14.25);  t.add(15.15);  t.add(18.25);  System.***out***.println("Initially Ascend-" + t);  System.***out***.println("Initially Descend-" + t.descendingSet());  System.***out***.println("ceiling(10.00)-" + t.ceiling(10.00));  System.***out***.println("higher(10.00)-" + t.higher(10.00));  System.***out***.println("floor(10.00)-" + t.floor(10.00));  System.***out***.println("lower(10.00-" + t.lower(10.00));  System.***out***.println("pollFirst()-" + t.pollFirst());  System.***out***.println("pollLast()-" + t.pollLast());  System.***out***.println("Now ascending Set-" + t);  System.***out***.println("Now descending Set-" + t.descendingSet());  }  } |
| Initially Ascend-[0.3, 1.45, 2.3, 4.2, 5.3, 5.45, 7.3, 9.2, 10.15, 12.25, 13.3, 14.25, 15.15, 18.25]  Initially Descend-[18.25, 15.15, 14.25, 13.3, 12.25, 10.15, 9.2, 7.3, 5.45, 5.3, 4.2, 2.3, 1.45, 0.3]  ceiling(10.00)-10.15  higher(10.00)-10.15  floor(10.00)-9.2  lower(10.00-9.2  pollFirst()-0.3  pollLast()-18.25  Now ascending Set-[1.45, 2.3, 4.2, 5.3, 5.45, 7.3, 9.2, 10.15, 12.25, 13.3, 14.25, 15.15]  Now descending Set-[15.15, 14.25, 13.3, 12.25, 10.15, 9.2, 7.3, 5.45, 5.3, 4.2, 2.3, 1.45] |

**NavigableMap**

It is very similar to NavigableSet what we have just discussed above. It is the child interface of SortedMap. It defines several methods for navigation purposes.

|  |  |  |  |
| --- | --- | --- | --- |
| Map(I) 🡸1.2V | SortedMap(I) 🡸1.2V | NavigableMap(I) 🡸1.6V | TreeMap (c) 1.2V |

**Methods:**

We have same number of methods here in NavigableMap also what we just had discussed in NavigableSet with just little difference as given below.

|  |  |
| --- | --- |
| **floorKey(10:00)** | Either at 10 or before 10 what is the last flight i.e. <=10:00 **🡺: Ans :- 9:20** |
| **floorKey(e)** | Returns highest element which is (less than or equal to) **<= e** |
|  | |
| **lowerKey(10:00)** | last flight before 10:00 O clock i.e. <10 🡺 **Ans: 9:20** |
| **lowerKey(e)** | Returns highest element which is less than e (< e) |
| **Note: Here we are heading towards lowest and we have to select highest from all lowest that is why we have used highest element in the above example.** | |
| **ceilingKey(10:00)** | the first flight either 10 or after 10:00 ‘O’ clock i.e. >=10 🡺 **Ans: 10:15** |
| **ceilingKey(e)** | Returns the lowest element which is (greater than or equal to e) > =e |
|  | |
| **higherKey(10:00)** | After 10:00 ‘O’ clock what is the first flight i.e. (greater than 10) > 10 🡺 **Ans : 10:15** |
| **higherKey(e)** | Returns the lowest element which greater than e (> e) |
| **Note: Here we are heading towards highest element (time) and we have to select lowest from all highest element that is why we have used lowest element in the above example.** | |
| pollFirstEntry() | Removes and returns first element |
| pollLastEntry() | Removes and returns last element |
| descendingMap() | It returns NavigableSet in reverse order |

So we have seen that the first four methods got appended with Key word because in map everything is done on Map key only and in the last two method Entry word got appended because here we have pair of (key-value) and which is nothing but an Entry in Map

|  |
| --- |
| **package** com.collection.navigableset.concept;  **import** java.util.TreeMap;  **public** **class** NavigableTreeMapConcept {  **public** **static** **void** main(String[] args) {  TreeMap<String, String> t = **new** TreeMap<String, String>();  t.put("b", "Banana");  t.put("c", "Cat");  t.put("a", "Apple");  t.put("d", "Dog");  t.put("g", "Gun");  System.***out***.println("Initially Ascend-" + t);  System.***out***.println("Initially Descend-" + t.descendingMap());  System.***out***.println("ceiling(e)-" + t.ceilingKey("e"));  System.***out***.println("higher(e)-" + t.higherKey("e"));  System.***out***.println("floor(e)-" + t.floorKey("e"));  System.***out***.println("lower(e)-" + t.lowerKey("e"));  System.***out***.println("pollFirst()-" + t.pollFirstEntry());  System.***out***.println("pollLast()-" + t.pollLastEntry());  System.***out***.println("Now ascending Set-" + t);  System.***out***.println("Now descending Set-" + t.descendingMap());  }  } |

|  |
| --- |
| **Output:**  Initially Ascend-{a=Apple, b=Banana, c=Cat, d=Dog, g=Gun}  Initially Descend-{g=Gun, d=Dog, c=Cat, b=Banana, a=Apple}  ceiling(e)-g  higher(e)-g  floor(e)-d  lower(e)-d  pollFirst()-a=Apple  pollLast()-g=Gun  Now ascending Set-{b=Banana, c=Cat, d=Dog}  Now descending Set-{d=Dog, c=Cat, b=Banana} |

Now we have covered all the topics from the collection except two utility classes.

**Collections** Class:

This is a utility class which contains several utility methods for Collection objects which is not available in other collection Interfaces and classes like (List, Set, & Queue). So fill that gap we have Collections utility class which defines several method like sorting, searching, reversing etc.

For example if we have to sorting for Set (I) then we have its implementation class TreeSet –class is there. If we have to do sorting for Queue then we have its implementation class PriorityQueue –class. But in case of List there is no any implementation class of list (ArrayList, LinkedList or Vector) who is doing sorting for List interface. So fill that gap we have sort () method in the Collections class like [Collections. sort (new ArrayList ()].

Sorting element of list ():

Collections class defines the following two sort() methods.

|  |  |
| --- | --- |
| **public static void sort(List l) :** | To sort based on DNSO. But in this case we need to fulfil two conditions   1. List should compulsory contain Homogeneous and comparable objects otherwise we will get RTE saying ClassCastException. 2. List should not contain null otherwise we will get NullPointerException while comparing a null with any elements |
| **public static void sort(List l, Comparator c)** | To sort based on Customized sorting order. |

**Demo Program for sorting list according to DNSO:**

|  |  |
| --- | --- |
| **import** java.util.ArrayList;  **import** java.util.Collections;  **public** **class** CollectionsArraListSort {  **public** **static** **void** main(String[] args) {  ArrayList al = **new** ArrayList();  al.add("Z");  al.add("A");  al.add("K");  al.add("N");  // al.add(1); // ClassCastException  // al.add(null);// NullPointerException  System.***out***.println("Before using sort()-" + al);  Collections.*sort*(al);  System.***out***.println("After using sort()-" + al);  }  } | Output:  Before using sort()-[Z, A, K, N]  After using sort()-[A, K, N, Z] |

Demo for customized sorting

|  |  |
| --- | --- |
| **package** com.collection.utilityclass.concept;  **import** java.util.Comparator;  **public** **class** MyComparator **implements** Comparator<Object> {  @Override  **public** **int** compare(Object obj1, Object obj2) {  **return** obj2.toString().compareTo(obj1.toString());  }  }  Output:  Before using sort()-[Z, A, K, N]  After using sort()-[Z, N, K, A] | **package** com.collection.utilityclass.concept;  **import** java.util.ArrayList;  **import** java.util.Collections;  **public** **class** CollectionsArraListSort {  **public** **static** **void** main(String[] args) {  ArrayList al = **new** ArrayList();  al.add("Z");  al.add("A");  al.add("K");  al.add("N");  // al.add(1); // ClassCastException  // al.add(null);// NullPointerException  System.***out***.println("Before using sort()-" + al);  **Collections.*sort*(al, new MyComparator());**  System.***out***.println("After using sort()-" + al);  }  } |

**Searching element of list:** How to search an element of a list or set or queue?

Collections class defines the following binarySearch () methods

1. **Public static int binarySearch(List l obj target):**

The binarySearch () method is based on Binary Search Algorithm and according to this algorithm if we have to search any element from the list then List must be sorted according to DNSO.

1. **Public static int binarySearch(List l, obj target,** **Comparator C):**

We have to use this method if the list is sorted according to customized sorting order

**Conclusion**:

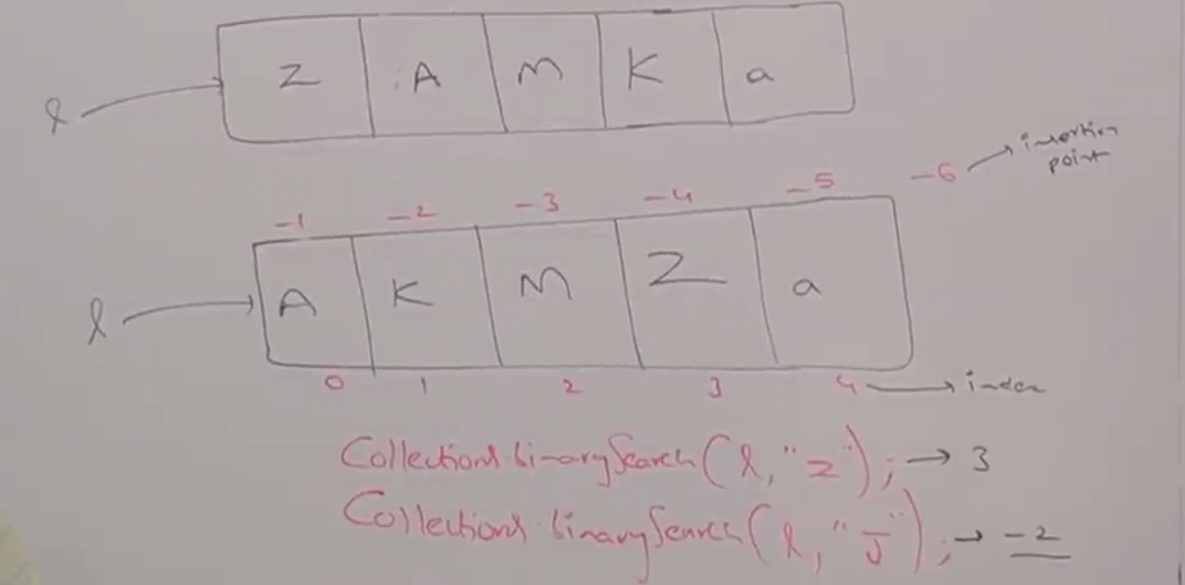
1. The above search method internally will use Binary Search Algorithm
2. Successful search returns index
3. Unsuccessful search returns insertion point
4. Insertion point is the location where we can place target element in sorted list
5. Before calling binarySearch() method compulsory list should be sorted otherwise we will get unpredictable results
6. If the list is sorted according to comparator then at the time search operation also we have to pass same comparator object otherwise we will get unpredictable results

|  |
| --- |
| **package com.collection.utilityclass.concept;**  **import java.util.ArrayList;**  **import java.util.Collections;**  **public class CollectionsArraListSort {**  **public static void main(String[] args) {**  **ArrayList al = new ArrayList();**  **al.add("Z");**  **al.add("A");**  **al.add("K");**  **al.add("N");**  **System.*out*.println("Before using sort()-" + al);**  **// Collections.sort(al);// For DNSO**  **Collections.*sort*(al, new MyComparator());// For Customized Sorting order**  **System.*out*.println("After using sort()-" + al);**  **// System.out.println(Collections.binarySearch(al, "B"));// For DNSO**  **System.*out*.println(Collections.*binarySearch*(al, "K", new MyComparator()));// For Customized Sorting order**  **}**  **}** |

|  |  |
| --- | --- |
| **package com.collection.utilityclass.concept;**  **import java.util.Comparator;**  **public class MyComparator implements Comparator<Object> {**  **@Override**  **public int compare(Object obj1, Object obj2) {**  **return obj2.toString().compareTo(obj1.toString());**  **}**  **}** | Before using sort()-[Z, A, K, N]  After using sort()-[Z, N, K, A]  2 |
| Let’s if we want to search the element which is not present in the sorted list then it will return insertion order like below | Before using sort()-[Z, A, K, N]  After using sort()-[Z, N, K, A]  -2 |

System.***out***.println(Collections.*binarySearch*(al, "P", **new** MyComparator()));// For Customized Sorting order

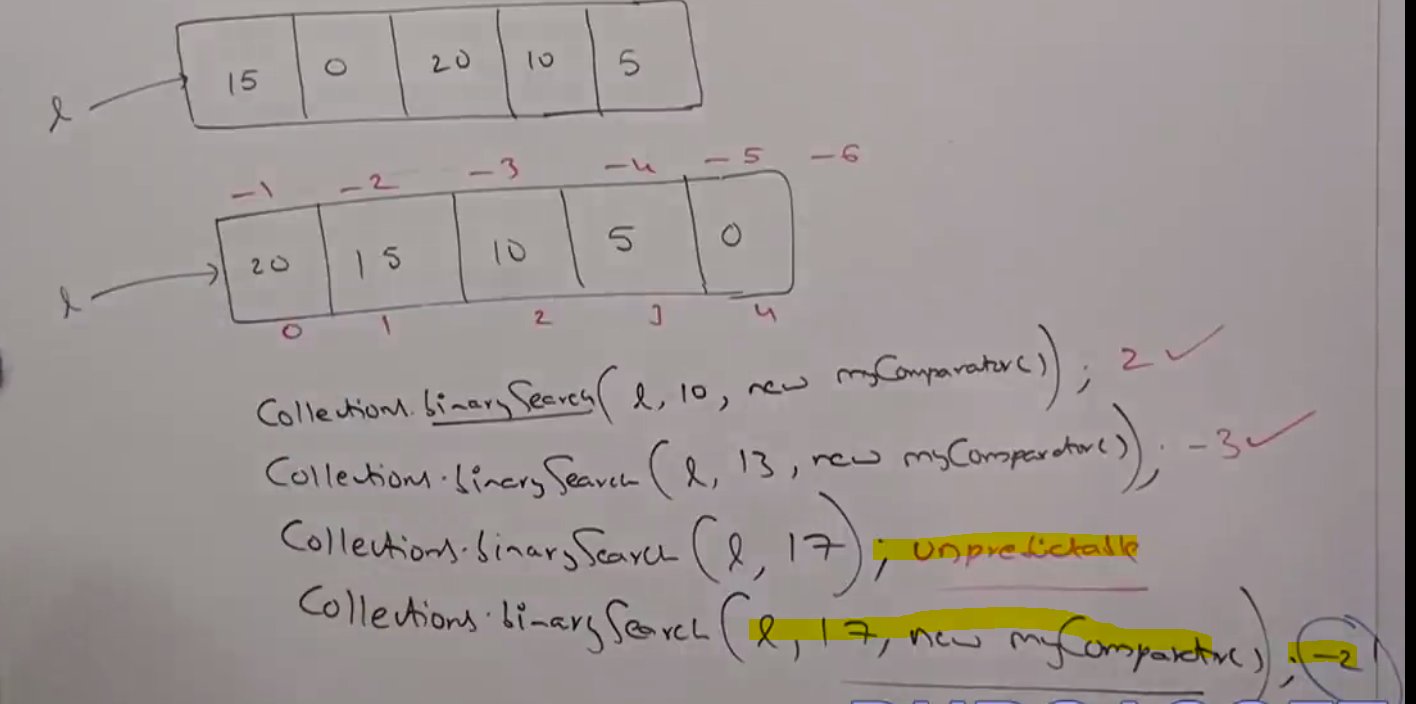
Here we are trying to search **P** which is not available in the list so we will get the insertion order of the P



Here in the number in (-) nothing but insertion order and number (+) is index number

|  |  |
| --- | --- |
| **package com.collection.utilityclass.concept;**  **import java.util.ArrayList;**  **import java.util.Collections;**  **public class CollectionsArraListSortNumber {**  **public static void main(String[] args) {**  **ArrayList al = new ArrayList();**  **al.add(10);**  **al.add(15);**  **al.add(9);**  **al.add(5);**  **al.add(2);**  **System.*out*.println("Before using sort()-" + al);**  **// Collections.sort(al);// For DNSO**  **Collections.*sort*(al, new MyComparator());// For Customized Sorting order**  **System.*out*.println("After using sort()-" + al);**  **// System.out.println(Collections.binarySearch(al, "B"));// For DNSO**  **System.*out*.println(Collections.*binarySearch*(al, 5, new MyComparator()));// For Customized Sorting order**  **}**  **}** | |
| **package com.collection.utilityclass.concept;**  **import java.util.Comparator;**  **public class MyComparator implements Comparator<Object> {**  **@Override**  **public int compare(Object obj1, Object obj2) {**  **// return obj2.toString().compareTo(obj1.toString());**  **return ((Integer) obj1).compareTo((Integer) obj2);**  **}**  **}** | Before using sort()-[10, 15, 9, 5, 2]  After using sort()-[2, 5, 9, 10, 15]  1  Output for searching 3  Before using sort()-[10, 15, 9, 5, 2]  After using sort()-[2, 5, 9, 10, 15]  -2 |

System.***out***.println(Collections.*binarySearch*(al, 3, **new** MyComparator()));// For Customized Sorting



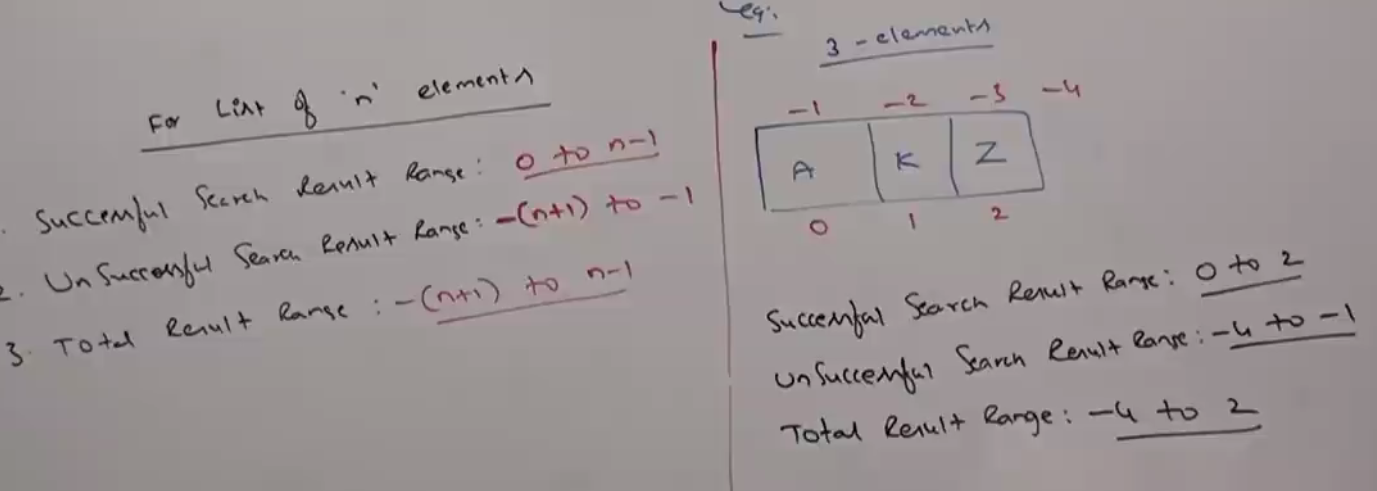
**Note:**

For the list of n elements in the case of binarySearch() method

1. Successful search result range is 0 to (n-1)
2. Unsucessful search result range is –(n+1) to -1
3. Total result range is –(n+1) to (n-1)

For example for three elements having elements [A, K,Z]

1. Successful search result range is 0 to (n-1)🡺 0 to (3-1)🡺 0 to +2
2. Unsucessful search result range is –(n+1) to -1🡺 -4 to -1 🡺 -(3+1) to -1 🡺-4 to -1
3. Total result range is –(n+1) to (n-1) 🡺 -(3+1) to (3-1) 🡺 -4 to +2



**Reversing elements of list-**

Collections class defines the following reverse method to reverse elements of list

|  |
| --- |
| **public static void reverse(list l)** |

Example:

|  |  |
| --- | --- |
| **package** com.collection.utilityclass.concept;  **import** java.util.ArrayList;  **import** java.util.Collections;  **public** **class** CollectionsArraListSortNumber {  **public** **static** **void** main(String[] args) {  ArrayList al = **new** ArrayList();  al.add(10);  al.add(15);  al.add(9);  al.add(5);  al.add(2);  System.***out***.println("Before reversal-" + al);  Collections.*reverse*(al);  System.***out***.println("After reversal-" + al);  }  } | Output:  Before reversal-[10, 15, 9, 5, 2]  After reversal-[2, 5, 9, 15, 10] |

**reverse vs. reverseOrder()**

We can use reverse method to reverse order of element of list, where as we can use reverseOrder() method to reversed Comparator.

|  |
| --- |
| Comparator c1 = Collections.reverseOrder(Comparator c)  Example:  Comparator<Object> c1 = Collections.*reverseOrder*(**new** MyComparator()); |

|  |
| --- |
| **package com.collection.utilityclass.concept;**  **import java.util.ArrayList;**  **import java.util.Collections;**  **import java.util.Comparator;**  **public class CollectionsArraListSortNumber {**  **public static void main(String[] args) {**  **ArrayList al = new ArrayList();**  **al.add(10);**  **al.add(15);**  **al.add(9);**  **al.add(5);**  **al.add(2);**  **System.*out*.println("Before sort-" + al);**  **Collections.*sort*(al, new MyComparator());**  **System.*out*.println("Before converting into reversal comparator & used sort-" + al);**  **Comparator<Object> c1 = Collections.*reverseOrder*(new MyComparator());**  **Collections.*sort*(al, c1);**  **System.*out*.println("After converting into reversal comparator-" + al);**  **}**  **}** |
| **package com.collection.utilityclass.concept;**  **import java.util.Comparator;**  **public class MyComparator implements Comparator<Object> {**  **@Override**  **public int compare(Object obj1, Object obj2) {**  **// return obj2.toString().compareTo(obj1.toString());**  **return ((Integer) obj2).compareTo((Integer) obj1);**  **}**  **}** |
| **Output:**  Before sort-[10, 15, 9, 5, 2]  Before converting into reversal comparator & used sort-[15, 10, 9, 5, 2]  After converting into reversal comparator-[2, 5, 9, 10, 15] |