Day-15 JA-111 Batch

Topics

- **The java.util.SortedSet<E> & java.util.NavigableSet<E> interface**
- **The java.util.TreeSet<E> class**
- **The java.util.Comparable<E> and java.util.Comparator<E> interface**
- **The PriorityQueue<E> class**
- **The java.util.Map<K, V> interface**
- **The java.util.HashMap<K,V> & java.util.LinkedHashMap<K,V> class**
- **The java.util.SortedMap<K,V> & java.util.NavigableMap< K,V> interface**
- **The java.util.TreeMap<K,V> class**
- **The java.util.HashTable<K.V> class**

The java.util.SortedSet<E> Interface

It is a subinterface of Set interface that maintains provides a total ordering on its elements according to a natural ordering or a comparator provided at the time of creation of SortedSet. It has following methods

Method	Description
Comparator<? super E> comparator()	Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
E first()	Returns the first (lowest) element currently in this set.
SortedSet<E> headSet(E toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
E last()	Returns the last (highest) element currently in this set.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet<E> tailSet(E fromElement)	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

The java.util.NavigableSet<E> Interface

It is a subinterface of SortedSet that provide easy navigation over Collection object. It allow traversal of Collection object in both direction. A NavigableSet interface provides following methods

Method	Description
E ceiling(E e)	Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
Iterator<E> descendingIterator()	Returns an iterator over the elements in this set, in descending order.
NavigableSet<E> descendingSet()	Returns a reverse order view of the elements contained in this set.

E floor(E e)	Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
SortedSet<E> headSet(E toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
NavigableSet<E> headSet(E toElement, boolean inclusive)	Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement.
E higher(E e)	Returns the least element in this set strictly greater than the given element, or null if there is no such element.
Iterator<E> iterator()	Returns an iterator over the elements in this set, in ascending order.
E lower(E e)	Returns the greatest element in this set strictly less than the given element, or null if there is no such element.
E pollFirst()	Retrieves and removes the first (lowest) element, or returns null if this set is empty.
E pollLast()	Retrieves and removes the last (highest) element, or returns null if this set is empty.
NavigableSet<E> subSet(E fromElement, booleanfromInclusive, E toElement, booleantoInclusive)	Returns a view of the portion of this set whose elements range from fromElement to toElement.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet<E> tailSet(E fromElement)	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
NavigableSet<E> tailSet(E fromElement, boolean inclusive)	Returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement.

Note: A class can implement the ***java.util.Comparable<E>*** interface to define the natural ordering of the objects. for example if you take a list of Strings, generally it is ordered by alphabetical comparisons. So when a String class is created, it can be made to implement Comparable interface and override the compareTo() method to provide the comparison definition. We can use them as,

```
str1.compareTo(str2);
```

Classes Implementing Comparable interface and their natural orders

Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical

Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
String	Lexicographic
LocalDate	Chronological

The java.util.TreeSet<E> class

- The TreeSet uses an red-black tree structure for storage of elements.
- It implements Set interface and stores element in ascending order.
- This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

The TreeSet provides following constructor

Constructor	**Description**
TreeSet()	Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
TreeSet(Collection<? extends E> c)	Constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.
TreeSet(Comparator<? super E> comparator)	Constructs a new, empty tree set, sorted according to the specified comparator.
TreeSet(SortedSet<E> s)	Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

An Example

```
package com.masai.sprint3;

import java.time.LocalDate;
import java.util.Set;
import java.util.TreeSet;

public class TreeSetDemo {
  public static void main(String[] args) {
    LocalDate physicsExamDate = LocalDate.parse("2022-12-08");
    LocalDate chemistryExamDate = LocalDate.parse("2022-12-10");
    LocalDate mathsExamDate = LocalDate.parse("2022-12-06");
    LocalDate englishExamDate = LocalDate.parse("2022-12-04");
    LocalDate PEExamDate = LocalDate.parse("2022-12-12");
```

```
    Set<LocalDate> dateSet = new TreeSet<>();
    dateSet.add(physicsExamDate);
    dateSet.add(chemistryExamDate);
    dateSet.add(mathsExamDate);
    dateSet.add(englishExamDate);
    dateSet.add(PEExamDate);

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

Output

```
[2022-12-04, 2022-12-06, 2022-12-08, 2022-12-10, 2022-12-12]
```

The java.util.Comparator<E> interface

As we have already discussed in the previous section that TreeSet allow us to store element in sorted order. By default elements are stored in natural ordering. A collection object may contain objects of user defined class and to apply a user defined ordering on them, java provides us facility to define criteria of sorting. To do so java has one more interface that is Comparator. Comparator interface have following methods

Method	**Description**
int compare(T o1, T o2)	Compares its two arguments for order. Returns: ➢ A negative integer if first argument is less than the second. ➢ Zero if first argument is equals the second. ➢ A positive integer if first argument is greater than the second.
boolean equals(Object obj)	Indicates whether some other object is "equal to" this comparator.

An Example

```
package com.masai.sprint3;

public class Mobile {
  private String modelNumber;
  private Integer price;

  public Mobile(String modelNumber, Integer price){
    this.modelNumber = modelNumber;
    this.price = price;
  }

  public void setModelNumber(String modelNumber){
    this.modelNumber = modelNumber;
  }

  public String getModelNumber(){
    return modelNumber;
```

```
  }

  public Integer getPrice(){
    return price;
  }

  public void setPrice(Integer price){
    this.price = price;
  }

  @Override
  public String toString(){
    return "Mobile [Model Number " + modelNumber + ", price = " + price
+ "]";
  }
}

package com.masai.sprint3;
import java.util.Comparator;

public class MobileSort implements Comparator<Mobile>{
  public int compare(Mobile m1, Mobile m2){
    if(m1.getPrice() > m2.getPrice())
      return 1;
    else if(m1.getPrice() < m2.getPrice())
      return -1;

    return m1.getModelNumber().compareTo(m2.getModelNumber());
  }
}

package com.masai.sprint3;
import java.util.Set;
import java.util.TreeSet;

public class TreeSetUserDefined {
  public static void main(String args[]){
    //Create three different Mobile objects
    Mobile mobileOne = new Mobile("A31", 22000);
    Mobile mobileTwo = new Mobile("Z31", 55000);
    Mobile mobileThree = new Mobile("S31", 34000);
    Mobile mobileFour = new Mobile("J33", 34000);

    //Create an object of TreeSet
    Set<Mobile> mobileSet = new TreeSet<>(new MobileSort());
    mobileSet.add(mobileOne);
    mobileSet.add(mobileTwo);
    mobileSet.add(mobileThree);
    mobileSet.add(mobileFour);
```

```
    for(Mobile temp: mobileSet){
      System.out.println(temp);
    }
  }
}
```

Output

```
Mobile [Model Number A31, price = 22000]
Mobile [Model Number J33, price = 34000]
Mobile [Model Number S31, price = 34000]
Mobile [Model Number Z31, price = 55000]
```

Difference between Comparable and Comparator interface

Comparable	Comparator
Comparable interface belongs to java.lang package	Comparator interface belongs to java.util package
If we want to specify the sorting logic of a class object within the same class , we need to use Comparable	If we want to specify the sorting logic of a class object outside that class then we should use Comparator.
With the help of Comparable we can define only one sorting logic within a class	With the help of Comparator we can define multiple sorting logic of a class object inside multiple classes.
Here method name is: public int compareTo(Object obj)	Here method name is: public int compare(Object obj1, Object obj2)

The java.util.PriorityQueue<E> class

- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time
- A priority queue does not permit null elements.
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily.

Constructor	Description
PriorityQueue()	Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
PriorityQueue(int initialCapacity)	Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
PriorityQueue(Comparator<? super E> comparator)	Creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)	Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

PriorityQueue(Collection<? extends E> c)	Creates a PriorityQueue containing the elements in the specified collection. If the specified collection is an instance of a SortedSet or is another PriorityQueue, this priority queue will be ordered according to the same ordering. Otherwise, this priority queue will be ordered according to the natural ordering of its elements.
PriorityQueue(PriorityQueue<? extends E> c)	Creates a PriorityQueue containing the elements in the specified priority queue. This priority queue will be ordered according to the same ordering as the given priority queue.
PriorityQueue(SortedSet<? extends E> c)	Creates a PriorityQueue containing the elements in the specified sorted set. This priority queue will be ordered according to the same ordering as the given sorted set

This class does not have any methods of its own. All methods of Collection interface and Queue interface are available

An Example

```
//filename: CustomerOrder.java
package com.masai.sprint3;

public class CustomerOrder implements Comparable<CustomerOrder> {
  private int orderId;
  private double orderAmount;
  private String customerName;

  public CustomerOrder(int orderId, double orderAmount, String customerName) {
    this.orderId = orderId;
    this.orderAmount = orderAmount;
    this.customerName = customerName;
  }

  @Override
  public int compareTo(CustomerOrder o) {
    if(o.orderAmount > this.orderAmount)
      return 1;
    else if (o.orderAmount < this.orderAmount)
      return -1;
    return o.orderId < this.orderId ? 1 : -1;
  }

  @Override
  public String toString() {
    return "orderId:" + this.orderId + ", orderAmount:" + this.orderAmount + ", customerName:" + customerName;
  }
```

```
  public int getOrderId() {
    return orderId;
  }

  public void setOrderId(int orderId) {
    this.orderId = orderId;
  }

  public String getCustomerName() {
    return customerName;
  }

  public void setCustomerName(String customerName) {
    this.customerName = customerName;
  }

  public void setOrderAmount(double orderAmount) {
    this.orderAmount = orderAmount;
  }

  public double getOrderAmount() {
      return orderAmount;
  }
}

//filename: PriorityQueueDemo.java
package com.masai.sprint3;
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueDemo{
  public static void main(String args[]){
    CustomerOrder c1 = new CustomerOrder(1, 100.0, "Aman");
    CustomerOrder c2 = new CustomerOrder(3, 50.0, "Chetan");
    CustomerOrder c3 = new CustomerOrder(2, 300.0, "Bhagat");

    Queue<CustomerOrder> customerOrders = new PriorityQueue<>();
    customerOrders.add(c1);
    customerOrders.add(c2);
    customerOrders.add(c3);
    while (!customerOrders.isEmpty())
      System.out.println(customerOrders.poll());
  }
}

Output
orderId:2, orderAmount:300.0, customerName:Bhagat
orderId:1, orderAmount:100.0, customerName:Aman
```

```
orderId:3, orderAmount:50.0, customerName:Chetan
```

Say Now you want to store these customer orders in the ascending order of customer name and in priority queue duplicate elements are allowed such that sorting order is decided arbitrarily. Let us write a Comparator that sorts customer order according to the name of customer only.

```
//filename: CustomerOrderByName.java
public class CustomerOrderByName implements Comparator<CustomerOrder> {
  @Override
  public int compare(CustomerOrder o1, CustomerOrder o2) {
    return o1.getCustomerName().compareTo(o2.getCustomerName());
  }
}

& add following code in the main method of PriorityQueueDemo

System.out.println("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=");

CustomerOrder c4 = new CustomerOrder(4, 50.0, "Chetan");
customerOrders = new PriorityQueue<>(new CustomerOrderByName());
customerOrders.add(c1);
customerOrders.add(c2);
customerOrders.add(c3);
customerOrders.add(c4);
while (!customerOrders.isEmpty())
  System.out.println(customerOrders.poll());

The output with updated main is:
orderId:2, orderAmount:300.0, customerName:Bhagat
orderId:1, orderAmount:100.0, customerName:Aman
orderId:3, orderAmount:50.0, customerName:Chetan
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
orderId:1, orderAmount:100.0, customerName:Aman
orderId:2, orderAmount:300.0, customerName:Bhagat
orderId:4, orderAmount:50.0, customerName:Chetan
orderId:3, orderAmount:50.0, customerName:Chetan
```

The java.util.Map<K, V> interface

- Map interface is top in its own hierarchy. The Map interface ***does not have*** Collection interface as its super interface.
- Map interface allow store value corresponding to a key.
- A map cannot have a duplicate key and a key can point to maximum one value.
- Map interface supports three kinds of views that are
  1. A set of keys   2. A set of values   3. A set of key-values
- A Map cannot contain itself as a key while it is perfectly valid for a Map to have itself as a value.
- Some Map interface implementation applies restriction on type of keys like they do not allow use null as a key and allow a particular type of object to be used as a key.

Map interface have following methods

Method	Description
void clear()	Removes all of the mappings from this map (optional operation)
boolean containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>> entrySet()	Returns a Set view of the mappings contained in this map.
int hashCode()	Returns the hash code value for this map.
boolean equals(Object o)	Compares the specified object with this map for equality.
V get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean isEmpty()	Returns true if this map contains no key-value mappings.
Set<K> keySet()	Returns a Set view of the keys contained in this map.
V put(K key, V value)	Associates the specified value with the specified key in this map (optional operation).
void putAll(Map<? extends K,? extends V> m)	Copies all of the mappings from the specified map to this map (optional operation).
V remove(Object key)	Removes the mapping for a key from this map if it is present (optional operation).
int size()	Returns the number of key-value mappings in this map.
Collection<V> values()	Returns a Collection view of the values contained in this map.

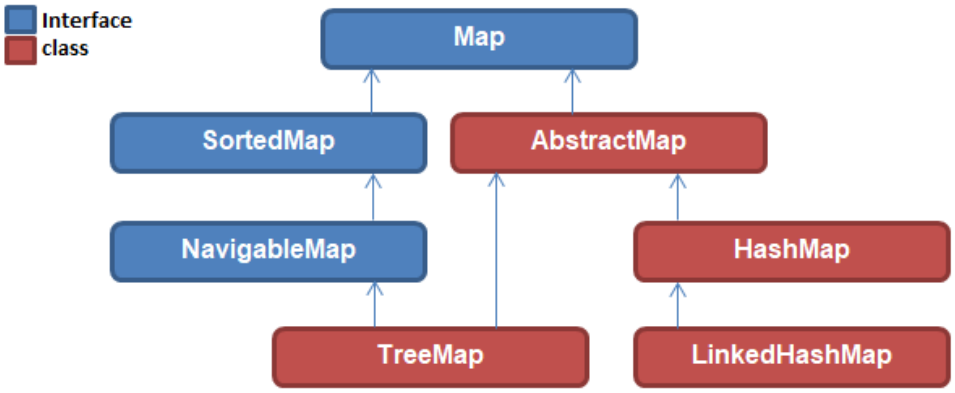
The java.util.Map.Entry<K,V> interface

It is a nested interface of Map interface. Map interface do not implement Iterable interface hence it is not possible to access all elements of Map interface sequentially. The Map.entrySet method returns a collection-view of the map, whose elements are of this class. The only way to obtain a reference to a map entry is from the iterator of this collection-view. These Map.Entry objects are valid only for the duration of the iteration, the behaviour of a map entry is undefined if the backing map has been modified after the entry was returned by the iterator, except through the setValue operation on the map entry. It has following methods

Method	Description
boolean equals(Object o)	Compares the specified object with this entry for equality.

K getKey()	Returns the key corresponding to this entry
V getValue()	Returns the value corresponding to this entry.
int hashCode()	Returns the hash code value for this map entry.
V setValue(V value)	Replaces the value corresponding to this entry with the specified value.

Hierarchy of Map interface and implementing classes



The java.util.HashMap<K,V> class

- HashMap class provides Hash table based implementation of Map interface.
- HashMap allow using null as key as well a value.
- Similar to HashSet class, HashMap also do not confirm order of elements according to their insertion order.

HashMap have following constructors

Method	**Description**
HashMap()	Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
HashMap(int initialCapacity)	Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
HashMap(int initialCapacity, float loadFactor)	Constructs an empty HashMap with the specified initial capacity and load factor.
HashMap(Map<? extends K,? extends V> m)	Constructs a new HashMap with the same mappings as the specified Map.

The HashMap class does not define any method of its own it just implement all methods of Map interface.

An Example:

```
import java.util.*;

package com.masai.sprint3;

import java.util.HashMap;
```

```
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class HashMapDemo{
  public static void main(String[] args) {
    Map<String,Integer> hm = new HashMap< >();
    hm.put("All", 3);
    hm.put("Is", 2);
    hm.put("Well", 3);
    System.out.println(hm);
    System.out.println("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=");

    hm.remove("Well");
    hm.put("Well!", 4);

    System.out.println("After Removal Map is : ");
    Set<Map.Entry<String, Integer>> st = hm.entrySet();
    Iterator<Map.Entry<String, Integer>> itr = st.iterator();
    while(itr.hasNext()){
      Map.Entry<String, Integer> entry = itr.next();
      System.out.print(entry.getKey() + ":" + entry.getValue() + " ");
    }
  }
}
```

Output of code given above is:

```
{All=3, Well=3, Is=2}
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
After Removal Map is :
All:3 Is:2 Well!:4
```

The java.util.LinkedHashMap<K,V> class

- LinkedHashMap class provide linked list and hash table implementation of Map interface.
- It also maintains order of element according to their insertion order.

LinkedHashMap provides following constructor

Constructor	**Description**
LinkedHashMap()	Constructs an empty insertion-ordered LinkedHashMap instance with the default initial capacity (16) and load factor (0.75).
LinkedHashMap(int initialCapacity)	Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and a default load factor (0.75).
LinkedHashMap(int initialCapacity, float loadFactor)	Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and load factor.

LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)	Constructs an empty LinkedHashMap instance with the specified initial capacity, load factor and ordering mode.
LinkedHashMap(Map<? extends K,? extends V> m)	Constructs an insertion-ordered LinkedHashMap instance with the same mappings as the specified map.

Just make one change in the above example

```
Map<String,Integer> hm = new HashMap<>();
```

Output of code given above is:

```
{All=3, Is=2, Well=3}
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
After Removal Map is :
All:3 Is:2 Well!:4
```

The java.util.SortedMap<K,V> interface

- SortedMap interface stores Map entries in sorted order. Elements are sorted according to natural ordering or according to a comparator that is provided at the time of SortedMap creation.
- All key that are inserted into SortedMap must implement comparable interface or Comparator has to be provided explicitly.

SortedMap interface have following methods

Method	**Description**
Comparator<? super K> comparator()	Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.
Set<Map.Entry<K,V>> entrySet()	Returns a Set view of the mappings contained in this map.
K firstKey()	Returns the first (lowest) key currently in this map.
SortedMap<K,V> headMap(K toKey)	Returns a view of the portion of this map whose keys are strictly less than toKey.
Set<K> keySet()	Returns a Set view of the keys contained in this map.
K lastKey()	Returns the last (highest) key currently in this map.
SortedMap<K,V> subMap(K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V> tailMap(K fromKey)	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
Collection<V> values()	Returns a Collection view of the values contained in this map.

The java.util.NavigableMap<K,V> interface

- It is a subinterface of SortedMap that provide easy navigation over Map object.
- It allow traversal of Map object in both direction.

A NavigableMap interface provides following methods

Method	Description
Map.Entry<K,V>ceilingEntry(K key)	Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
K ceilingKey(K key)	Returns the least key greater than or equal to the given key, or null if there is no such key.
NavigableSet<K>descendingKeySet()	Returns a reverse order NavigableSet view of the keys contained in this map.
NavigableMap<K,V>descendingMap()	Returns a reverse order view of the mappings contained in this map.
Map.Entry<K,V>firstEntry()	Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V>floorEntry(K key)	Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
K floorKey(K key)	Returns the greatest key less than or equal to the given key, or null if there is no such key.
SortedMap<K,V>headMap(K toKey)	Returns a view of the portion of this map whose keys are strictly less than toKey.
NavigableMap<K,V>headMap(K toKey, boolean inclusive)	Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.
Map.Entry<K,V>higherEntry(K key)	Returns a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.
K higherKey(K key)	Returns the least key strictly greater than the given key, or null if there is no such key.
Map.Entry<K,V>lastEntry()	Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
Map.Entry<K,V>lowerEntry(K key)	Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K lowerKey(K key)	Returns the greatest key strictly less than the given key, or null if there is no such key.
NavigableSet<K>navigableKeySet()	Returns a NavigableSet view of the keys contained in this map.
Map.Entry<K,V>pollFirstEntry()	Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V>pollLastEntry()	Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is

	empty.
NavigableMap<K,V>subMap(K fromKey, booleanfromInclusive, K toKey, booleantoInclusive)	Returns a view of the portion of this map whose keys range from fromKey to toKey.
SortedMap<K,V>subMap(K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V>tailMap(K fromKey)	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
NavigableMap<K,V>tailMap(K fromKey, boolean inclusive)	Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.

The java.util.TreeMap<K,V> interface

- A Red-Black tree based NavigableMap implementation.
- TreeMap class is used to store map entries sorted according to the natural ordering of keys, or according to a Comparator.

The TreeMap provides following constructor

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

An Example

```
//Student.java
package com.masai.sprint3;

public class Student  {
  private int rollNo;
  private String name;
  private int marks;

  public Student(int rollNo, String name, int mark) {
    this.rollNo = rollNo;
    this.name = name;
    this.marks = mark;
  }
```

```
  public int getRoll() {
    return rollNo;
  }

  public void setRoll(int roll) {
    this.rollNo = roll;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getMarks() {
    return marks;
  }

  public void setMarks(int marks) {
    this.marks = marks;
  }

     @Override
  public String toString(){
     return "Roll No: " + rollNo + " Name: " + name + " Marks: " +
marks;
  }
}

//StudentMarksComp.java
package com.masai.sprint3;
import java.util.Comparator;

public class StudentMarksComp implements Comparator<Student> {
  @Override
  public int compare(Student s1, Student s2) {
    if(s1.getMarks() > s2.getMarks())
      return 1;
    else if(s1.getMarks() < s2.getMarks())
      return 1;
    else
      return 0;
  }
}

//TreeMapDemo.java
package com.masai.sprint3;
```

```
import java.util.*;

public class TreeMapDemo {
  public static void main(String[] args) {
    TreeMap<Student,String> tm = new TreeMap<>(new StudentMarksComp());
    tm.put(new Student(10,"Ganesh",950),"Maharastra");
    tm.put(new Student(12,"Surya",850),"Tamilnadu");
    tm.put(new Student(15,"Venkat",920),"Telangana");
    tm.put(new Student(16,"Dinesh",910),"Haryana");
    tm.put(new Student(18,"Srinu",880),"Kerla");

    for(Map.Entry<Student,String> me: tm.entrySet()) {
      System.out.println("Toppers Student of State " + me.getValue() + "
is " + me.getKey());
    }
  }
}
```

Output

```
Toppers Student of State Maharastra is Roll No: 10 Name: Ganesh Marks:
950
Toppers Student of State Tamilnadu is Roll No: 12 Name: Surya Marks: 850
Toppers Student of State Telangana is Roll No: 15 Name: Venkat Marks:
920
Toppers Student of State Haryana is Roll No: 16 Name: Dinesh Marks: 910
Toppers Student of State Kerla is Roll No: 18 Name: Srinu Marks: 880
```

The java.util.Hashtable<K,V> class

- HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.
- But there are following main differences between HashMap and Hashtable class:
  1. HashMap is non-synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code. whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
  2. HashMap allows one null key and multiple null values whereas Hashtable doesn't allow any null key or value.

An Example

```
package com.masai.sprint3;
import java.util.*;

public class HashTableDemo{
  public static void main(String args[]){
    Hashtable<Integer,String> hm = new Hashtable<Integer,String>();
    hm.put(100,"Amit");
    hm.put(102,"Ravi");
    hm.put(101,"Vijay");
    hm.put(103,"Rahul");
```

```
    for(Map.Entry<Integer,String>  m:hm.entrySet())
      System.out.println(m.getKey()+"  "+m.getValue());
  }
}
```

Output

```
103  Rahul
102  Ravi
101  Vijay
100  Amit
```