Master Java Collections Framework

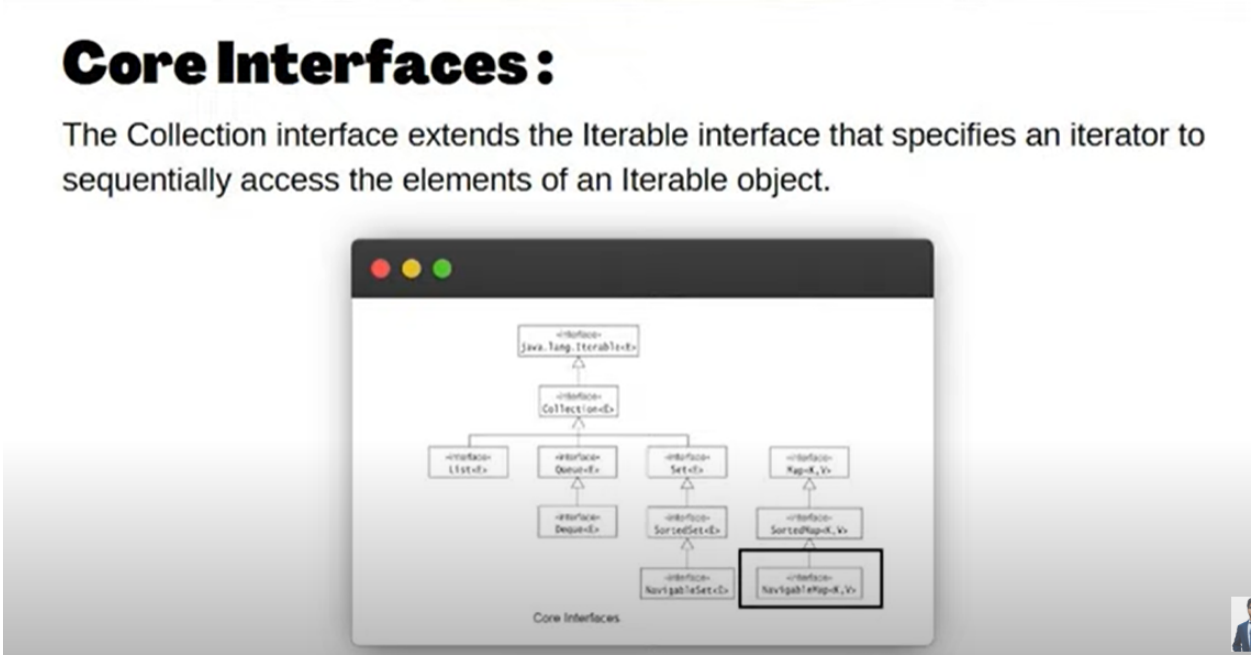
By- Riddhi Dutta

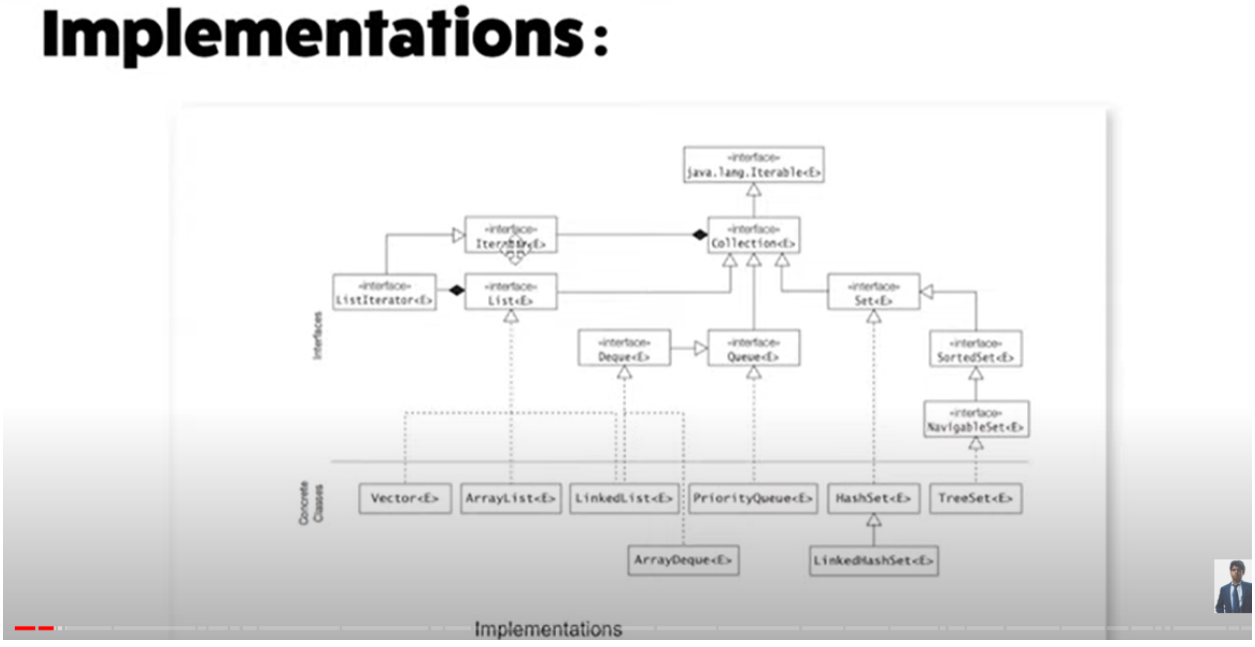
Video link- https://www.youtube.com/watch?v=VE_AAUxTUCY&t=6s
Made by -Priya Pramanick
Linkedin- https://www.linkedin.com/in/priya-pramanick-06b41121b/
Twitter - https://twitter.com/priya_nick_17
GitHub-https://github.com/priya-172

Topics :

- **Collections Core Interfaces**
- **Collections Concrete Classes**
- **Need of Iterators**
- **Iterable and Iterator Interface**
- **Collection Interface**
- **Lists**
- **ArrayList**
- **The Vector Class in Java**
- **The LinkedList Class in Java**
- **ArrayList vs LinkedList**
- **List Code Demo and Methods**
- **List Iterators**
- **Queue Interface in Java**
- **Deque Interface**
- **Queue Code Demo and Important Methods**
- **Stack Class in Java**
- **ArrayDeque in Java**
- **Priority Queue in Java**
- **The Comparable Interface**
- **The Comparator Interface**
- **The Set Interface in Java , HashSet**
- **LinkedHashSet**
- **Internal Working of HashSet , equals() and hashCode()**
- **SortedSet Interface**
- **NavigableSet Interface**
- **TreeSet in Java**
- **The Map Interface in Java**
- **HashMaps and HashTables**
- **Creating an adjacency list for graph questions using SortedMap Interface**
- **NavigableMap Interface**
- **TreeMap in Java**
- **Sorting an array and a list**
- **Some more important methods of Collections and Arrays**

Collections Core Interfaces





Need of Iterators

- Iterators provide a way to traverse and access elements in a collection sequentially without exposing the underlying implementation details. It ensures a

consistent and uniform way of accessing elements across different collection types.

- ❖
- ❖ Iterators allow safe removal of elements during iteration. They provide a remove() method to remove elements from the collection while iterating, avoiding potential concurrent modification exceptions.
- ❖
- ❖ Iterators enable the enhanced for loop syntax in Java, making it easy to iterate over collections without explicitly managing the loop counter or index.
- ❖
- ❖ Iterators provide a fail-fast behavior, which means they throw a ConcurrentModificationException if the collection is modified while iterating. This helps in detecting and handling potential concurrent modification issues.
- ❖
- ❖ Iterators support different types of traversals, such as forward, backward, and filtered traversals, depending on the specific iterator implementation. This flexibility allows developers to choose the most appropriate traversal method for their needs.

Iterable Interface

OurGenericsList.java

```
package priya;

import java.util.Iterator;

public class OurGenericsList<T> implements Iterable<T>{
private T[] items;
private int size;
public OurGenericsList(){
    size=0;
    items= (T[])new Object[100];

}
public void add(T item){
    items[size++]=item;
}
public T getItemAIndex(int index){
    return items[index];
}
@Override
```

```
public Iterator<T> iterator() {
   return new OurGenericsListIterator(this);
}
private class OurGenericsListIterator implements Iterator<T>{
    private OurGenericsList<T> list;
    private int index=0;
    public OurGenericsListIterator(OurGenericsList<T> list){
        this.list=list;
    }
    @Override
    public boolean hasNext() {
        System.out.println("hasnext called");
      return index < list.size;
    }
    @Override
    public T next() {
        System.out.println("next called");
     return list.items[index++];
    }
}

}
```

CollectionTest.java

```
package priya;

import java.util.Iterator;

public class CollectionTest {
    public static void main(String[] args) {
        OurGenericsList<Integer> list= new OurGenericsList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        // Iterator<Integer> iterator = list.iterator();
        // while(iterator.hasNext())
        // System.out.println(iterator.next());
        for(int x : list)
            System.out.println(x);
    }
```

```
}
```

Output
hasnext called
next called
1
hasnext called
next called
2
hasnext called
next called
3
hasnext called

ArrayList

```
package priya;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class CollectionTest {
    public static void main(String[] args) {
    //creating ArrayList
    List<Integer> alist =new ArrayList<>();
        alist.add(1);
        alist.add(2);
        alist.add(3);
        //to replaced a elemnt
       // System.out.println("element that got
replaced="+alist.set(1,100));//alist.set(index,element)
        System.out.println(alist);
        //coping all content in alist to alist2
        //List<Integer> alist2 =new ArrayList<>(alist);
        //System.out.println(alist2);//[1, 2, 3]
        List<Integer> alist2 =new ArrayList<>();
```

```
        alist2.add(4);
        alist2.add(5);
        alist2.add(6);
        alist2.addAll(alist);
        System.out.println(alist2);//[4, 5, 6, 1, 2, 3]
        //if we want to get particular index of an element
        System.out.println(alist);
        System.out.println(alist.indexOf(2));//1
         //sublist
        List<Integer> alist3 = alist2.subList(1,4); //fromIndex->including
, toIndex->excluding
        System.out.println(alist3);   //[5, 6, 1]
        //if we change the sublist it will change the ArrayList
too-->sublist method is copy of reference
        alist3.set(0,100);
        System.out.println(alist3);//[100, 6, 1]
        System.out.println(alist2);// [4, 100, 6, 1, 2, 3]
    }}
```

ListIterator

Interface Lisiterator<E> extends iterator<E>
{
boolean hasNext();
boolean hasPrevious();
E next();//Element after the cursor
E previous();//Element before the cursor
}

- ➢ List provides two methods
  - ListIterator<E>listIterator()
  - ListIterator<E>listIterator(index)
- ➢ The ListIterator interface is a bidirectional iterator for lists.
- ➢ It extends the iterator interface and allows the list to traversed in either direction using next() and prev().

```
package priya;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
```

```java
import java.util.ListIterator;

public class CollectionTest {
    public static void main(String[] args) {
    //creating ArrayList
    List<Integer> lk=new LinkedList<>();
    lk.add(1);
    lk.add(2);
    lk.add(3);
    ListIterator<Integer> iterator= lk.listIterator();
    //next->returning the current element then go to the next
element
    //next= return items[index++];
    System.out.println(iterator.next());//1
    System.out.println(iterator.next());//2
    //previous will first move back the pointer then return the
element
    //prev=return items[--index];
    System.out.println(iterator.previous());//2
    }
}
```

Better to use a Arraylist

ArrayList to Array for Leetcode

```java
package priya;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class CollectionTest {
    public static void main(String[] args) {
   //Arralist to array for leetcode
   //creating a ArrayList
   List<Integer> alist =new ArrayList<>();
   alist.add(1);
   alist.add(2);
   alist.add(3);
```

```
   alist.add(4);
   //ArraList to array
   //if array.size()<ArrayList.size()->new array will be created
which size will be equivalent to ArrayList size
   //if array.size()>ArrayList.size()->then extra index of array
will be filled with null value
   Integer [] arr=alist.toArray(new Integer[0]);
   //autoboxing,unboxing
   for(int x: arr) System.out.print(x+",");//1,2,3,4,

    }
}
```

Queues

The Queues interface extends the collection interface with the following methods:

- boolean add(E element)
- boolean offer(E element)//better to use we don't get unwanted exception. The pole method doesn't show any exception and return null
- E poll()
- E peek()//better to use peek element we don't get unwanted exception
- E element()

Deque:

- The Deque interface implements the Queue interface to allow double-ended queues.
- It allows operations not just at its head,but also at its tail.
- Element can be inserted at or removed from either end.
- A deque can be used as FIFO queue,where elements added at the tail are presented at the head for inspection or removal in the same order thus implementation FIFO order.

Adding elements

- boolean offerFirst(E element)
- boolean offerLast(E element) Queue equivalent: offer()
- Void push(E element) synonym: addFirst()
- Void addFirst(E element)
- Void addLast(E element) queue equivalent: add()

Removing elements

- E pollFirst() Queue equivalent: poll()
- E pollLast()
- E pop() synonym: removeFirst()
- E removeFirst() Queue equivalent: remove()
- E removeLast()

Examine

- E peekFirst() Queue equivalent: peek()
- E peekLast()
- E getFirst() Queue equivalent: element()
- E getLast()

When you want to do FIFO go for Araaydeque.
Whenever we need a normal queue we would use Linkedlist.
Whenever we need a stack we would use Array deque.

```
package priya;

import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.Queue;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
//creating a FIFO queue
Queue<Integer> q= new LinkedList<>();
q.offer(1);
q.offer(2);
System.out.println(q.peek());//1
System.out.println(q.poll());//1
System.out.println(q.peek());//2
System.out.println(q.isEmpty());//false
```

```
//stack
Stack<Integer> s = new Stack<>();
s.push(1);
s.push(2);
while(!s.isEmpty()){
System.out.println(s.peek());//2,1
s.pop();
}
//ArrayDque
Deque<Integer> dq = new ArrayDeque<>();
dq.offer(1);
dq.offer(2);
System.out.println(dq);//[1,2]
dq.pollFirst();
dq.peekFirst();
System.out.println(dq);//[2]

    }
}
```

Priority Queue

- It works based on the priority we fixed.
- The implementation is based on a priority heap, a tree like structure that yields an element at the head of the queue according ti priority ordering , which is defined either by the natural ordering of its element by comparator.
- In the case of several elements having same priority, ome of them is chosen arbitrarily.
- Elements of a PriorityQueue are not sorted.
- The queue only guarantees that elements can be removed in priority order and any traversal using an iterator does not.

Comparable

StudentMarks.java

```
package priya;
public class StudentMarks implements Comparable<StudentMarks> {
    private  int maths;
    private int physics;
    public String toString(){
return "StudentMarks[maths="+ maths +",physics="+physics+"]";
    }
```

```
    public int getMaths(){
        return maths;
    }
    public int getPhysics(){
        return physics;
    }
    public StudentMarks(int maths, int physics){
        this.maths=maths;
        this.physics=physics;
    }
    @Override
    public int compareTo(StudentMarks o) {
    // if(this.maths< o.maths) return -1;
    // if(this.maths> o.maths) return 1;
    // if(this.maths==o.maths) return 0;
    System.out.println("compare to() is called");
    return o.maths-this.maths;
    }
}
```

CollectionTest.java

```
package priya;

import java.util.ArrayList;
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
//PriorityQueue with comparable interface
//get the marks of top3 student
List<StudentMarks> stmarks= new ArrayList<>();
stmarks.add(new StudentMarks(70,80));
```

```
stmarks.add(new StudentMarks(50,60));
stmarks.add(new StudentMarks(30,50));
stmarks.add(new StudentMarks(45,88));
stmarks.add(new StudentMarks(90,100));
PriorityQueue<StudentMarks> spq=new PriorityQueue<>(stmarks);
System.out.println(spq);//[StudentMarks[maths=90,physics=100],
StudentMarks[maths=70,physics=80],
StudentMarks[maths=30,physics=50],
StudentMarks[maths=45,physics=88],
StudentMarks[maths=50,physics=60]]
List<StudentMarks> top3=new ArrayList<>();
int index=0;
while(!spq.isEmpty()){
    if(index==3) break;
    top3.add(spq.poll());
    index++;
}
System.out.println(top3);//[StudentMarks[maths=90,physics=100],
StudentMarks[maths=70,physics=80],
StudentMarks[maths=50,physics=60]]
    }
}
```

Comparator

Mycomparator.java

```
package priya;

import java.util.Comparator;

public class Mycomparator implements Comparator<Integer> {
@Override
public int compare(Integer o1, Integer o2) {
    return o2-o1;
}
}
```

CollectionTest.java

```java
package priya;

import java.util.ArrayList;
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
//PriorityQueue with comparator
PriorityQueue<Integer> pq = new PriorityQueue<>(new Mycomparator());
pq.offer(1);
pq.offer(2);
pq.offer(0);
pq.offer(100);
System.out.println(pq);//[100,2,0,1]
//top 2 elements
List<Integer> top2=new ArrayList<>();
int index=0;
while(!pq.isEmpty()){
if(index==2) break;
top2.add(pq.poll());
index++;
}
//System.out.println(pq);
System.out.println(top2);//[100,2]
System.out.println(pq);//[1,0]

    }
}
```

Lamda Function

```java
package priya;
```

```java
import java.util.ArrayList;
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
//riorityQueue using lamda functiom
PriorityQueue<Integer> pq = new PriorityQueue<>((a,b)->b-a);
pq.offer(1);
pq.offer(2);
pq.offer(0);
pq.offer(100);
System.out.println(pq);
//top 2 elements
List<Integer> top2=new ArrayList<>();
int index=0;
while(!pq.isEmpty()){
if(index==2) break;
top2.add(pq.poll());
index++;
}
//System.out.println(pq);
System.out.println(top2);
System.out.println(pq);
}
}
```

If we want do more action inside lamda function

```java
package priya;

import java.util.ArrayList;
```

```java
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
//priorityQueue using lamda functiom
//if we want to sort according to physics number and top3 among them
List<StudentMarks> stmarks= new ArrayList<>();
stmarks.add(new StudentMarks(70,80));
stmarks.add(new StudentMarks(50,60));
stmarks.add(new StudentMarks(30,50));
stmarks.add(new StudentMarks(45,88));
stmarks.add(new StudentMarks(90,100));
//if we want to do some action inside lamda function
PriorityQueue<StudentMarks> spq=new
PriorityQueue<StudentMarks>((s1,s2)->{
    System.out.println("comparator's compareTo() is called");
    return s2.getPhysics()-s1.getPhysics();
});
for(StudentMarks sm : stmarks) spq.add(sm);
List<StudentMarks> top3=new ArrayList<>();
int index=0;
while(!spq.isEmpty()){
    if(index==3) break;
    top3.add(spq.poll());
    index++;
}
System.out.println(top3);
//comparator's compareTo() is called
// comparator's compareTo() is called
// comparator's compareTo() is called
// comparator's compareTo() is called
```

```
// comparator's compareTo() is called
// comparator's compareTo() is called
// [StudentMarks[maths=90,physics=100],
StudentMarks[maths=45,physics=88],
StudentMarks[maths=70,physics=80]]
    }
}
```

Comparator is more flexible

Sets

Functions

- a.containsAll(b) (subset)
- a.addAll(b) (union)
- a.removeAll(b) (difference)
- a.retainAll(b) (intersection)
- a.clear() (empty set)

CollectionTest.java

```
package priya;

import java.util.ArrayList;
import java.util.Deque;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
//set
Set<Integer> set1=new HashSet<>();
Set<Integer> set2=new HashSet<>();
//Set<Integer> set=new HashSet<>(alist);//we acn add ArrayList to set
set1.add(2);//to add elemts set.add(elmeent)
```

```
set1.add(1);
set1.add(3);
set1.add(2);
//set.remove(2);//to add elemts set.add(elmeent)
//set.clear();//[]
set2.add(2);//to add elemts set.add(elmeent)
set2.add(5);
set2.add(3);
set2.add(4);
//intersection of two sets
System.out.println(set1);//[1,2,3]
System.out.println(set2);//[2,3,4,5]
System.out.println("after retaining");
set1.retainAll(set2);
System.out.println(set1);//[2,3]
System.out.println(set2);//[2,3,4,5]
//remove intersection of two sets
System.out.println(set1);//[1,2,3]
System.out.println(set2);//[2,3,4,5]
System.out.println("after retaining");
set1.removeAll(set2);
System.out.println(set1);//[1]
System.out.println(set2);//[2,3,4,5]
//union of two sets
System.out.println(set1);//[1,2,3]
System.out.println(set2);//[2,3,4,5]
System.out.println("after retaining");
set1.addAll(set2);
System.out.println(set1);//[1,2,3,4,5]
System.out.println(set2);//[2,3,4,5]
}
}
```

HashSet insertion order doesn't preserve so LinkesHashSet came into picture

```
package priya;

import java.util.ArrayList;
import java.util.Deque;
import java.util.HashSet;
```

```java
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
// HashSet
Set<Integer> set1=new HashSet<>();
//LinkedHashSet
Set<Integer> set2=new LinkedHashSet<>();
//adding elements in set1
set1.add(2);
set1.add(1);
set1.add(3);
set1.add(2);
//adding elements in set2
set2.add(2);
set2.add(5);
set2.add(3);
set2.add(4);
System.out.println(set1);//insertion order doesn't preserve[1,2,3]
System.out.println(set2);//insertion order preserve[2,5,3,4]
    }
}
```

HASHCODE

```java
package priya;

import java.util.ArrayList;
import java.util.Deque;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
```

```java
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.Stack;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
List<StudentMarks> stmarks= new ArrayList<>();
stmarks.add(new StudentMarks(70,80));
stmarks.add(new StudentMarks(50,60));
stmarks.add(new StudentMarks(30,50));
stmarks.add(new StudentMarks(45,88));
stmarks.add(new StudentMarks(90,100));
//set using custom class
Set<StudentMarks> set3 =new HashSet<>(stmarks);
System.out.println(set3.contains(new StudentMarks(70,80)));//false..why???
is it there in the HashSet
//whenever we are using HashSet with custom classes we must override
equals() and hashCode() before using it
System.out.println(set3.contains(new StudentMarks(70,80)));//trueee
    }
}
```

whenever we are using HashSet with custom classes we must override equals() and hashCode() before using it

StudentMarks.java

```java
package priya;
public class StudentMarks implements Comparable<StudentMarks> {
    private  int maths;
    private int physics;
    public String toString(){
return "StudentMarks[maths="+ maths +",physics="+physics+"]";
    }
```

```
    public int getMaths(){
        return maths;
    }
    public int getPhysics(){
        return physics;
    }
    public StudentMarks(int maths, int physics){
        this.maths=maths;
        this.physics=physics;
    }
    @Override
    public int compareTo(StudentMarks o) {
    // if(this.maths< o.maths) return -1;
    // if(this.maths> o.maths) return 1;
    // if(this.maths==o.maths) return 0;
    System.out.println("compare to() is called");
    return o.maths-this.maths;
    }
    @Override
    public int hashCode(){
        final int prime=31;
        int res=1;
        res=prime*res+maths;
        res=prime*res+physics;
        return res;
    }
    @Override
    public boolean equals(Object obj){
        if(this==obj) return true;
        if(this==null) return false;
        if(getClass()!=obj.getClass()) return false;
        StudentMarks other=(StudentMarks) obj;
        if(maths!=other.maths) return false;
        if(physics!= other.physics) return false;
        return true;
    }

}
        //explanation
        //implement hashcode() and equals() of your own otherwise
```

```
        //those functions from object class will be used
        //where objects are compared
        //so when write new inside contains
        //a new object is created obviously won't be equal to object
already created before hand
        //so we write a new hashcode and equals implementation
        //where even if objects are different
        //if contents of object are same i.e. maths and physics marks
        //then true is returned which we want
```

Sorted Set

- The SortedSet interface extends the set interface to provide the functionality for handling sorted set.
- Since the elements are sorted, traversing the set either using the for(:) loop or an iterator will access the elements according to the ordering used by the set.
    - ❖ E first()
    - ❖ E last()

The NavigableSet interface

- Extends the SortedSet interface with navigation methods to find the closet matches for specific search targets.
- By navigation, we mean operations that require searching for elements in the navigable set.
- In the absence of elements, these operations return null rather than throw a NoSuchElementException.
- **NavigableSet** interface adds some new methods:

  //First-last elements //Closet-matches
    - E pollFirst() E ceiling() E higher(E e)
    - E pollLast() E floor(E e) E lower(E e)

TreeSet

```
public class CollectionTest {
    public static void main(String[] args) {
List<StudentMarks> stmarks= new ArrayList<>();
stmarks.add(new StudentMarks(70,80));
stmarks.add(new StudentMarks(50,60));
stmarks.add(new StudentMarks(30,50));
stmarks.add(new StudentMarks(45,88));
stmarks.add(new StudentMarks(90,100));
```

```
//TreeSet-->sorted according to maths number
Set<StudentMarks> treeSet =new TreeSet<>(stmarks);
for(StudentMarks x: treeSet) System.out.println(x+",");
                                        }
                                    }
compare to() is called
compare to() is called
compare to() is called
compare to() is called
compare to() is called
StudentMarks[maths=90,physics=100],
StudentMarks[maths=50,physics=60],
StudentMarks[maths=45,physics=88],
StudentMarks[maths=30,physics=50],
```

According to physics number

```
public class CollectionTest {
    public static void main(String[] args) {
List<StudentMarks> stmarks= new ArrayList<>();
//TreeSet-->sorted according to physics number
Set<StudentMarks> treeSet =new
TreeSet<>((s1,s2)->s2.getPhysics()-s1.getPhysics());
treeSet.add(new StudentMarks(70,80));
treeSet.add(new StudentMarks(50,60));
treeSet.add(new StudentMarks(30,50));
treeSet.add(new StudentMarks(45,88));
treeSet.add(new StudentMarks(90,100));
for(StudentMarks x: treeSet) System.out.println(x+",");
    }
}
//StudentMarks[maths=90,physics=100],
StudentMarks[maths=45,physics=88],
StudentMarks[maths=70,physics=80],
StudentMarks[maths=50,physics=60],
StudentMarks[maths=30,physics=50],
```

Floor and ceiling

```
public class CollectionTest {
    public static void main(String[] args) {
//floor of ceil function part of NavigableSet
```

```
NavigableSet<Integer> set5= new TreeSet<>();
set5.add(8);
set5.add(5);
set5.add(1);
set5.add(9);
for(int x:set5) System.out.println(x+",");
System.out.println(set5.floor(4));//1-->return lessthan or equal to
System.out.println(set5.higher(4));//5-->greater than 4
System.out.println(set5.lower(4));//1-->reater than 4
System.out.println(set5.ceiling(4));//5-->return greater than or equal to-
    }
}
```

MAP

- A Map defines mappings from keys to values.
- The <Key,value> pair is called as entry.
- A map does not allow duplicates keys, in other words, the keys are unique.
- Both the keys and values must be objects, with primitive values being wrapped in their respective primitive wrapper objects when they are put in a map.
- A map is not a collection and the map interface does not extend the collection interface.
- However, the mappings can be viewed as a collection in various ways: a key set, a value collection or any entry set.

Map interface methods:

- Object put(K key, V value)
- Object get(Object key)
- Object remove(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- Int size()
- boolean isEmpty()

Bulk Operations:

- Void putAll(Map<? Extends k, ? extends V>map)
- Void clear()
- Set<K>keySet()
- Collection<V> values()
- Set<Map.Entry<K,V>> entrySet()

Interface Entry<K,V>{
K getKey();

V getValue();
V setValue(V value);
}

- Each <Key,value> in the entry set view is represented by an object implementing nested set.
- An entry in the entry set view can be manipulated by methods defined in this interface.

Implementation

```
package priya;

import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;
import java.util.NavigableMap;
import java.util.NavigableSet;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.Stack;
import java.util.TreeSet;

import javax.swing.border.EmptyBorder;

public class CollectionTest {
    public static void main(String[] args) {
//creating Map
Map<String,Integer> mp=new HashMap<>();
mp.put("priya",1);
mp.put("riya",2);
mp.put("pri",3);
//mp.remove("priya");
```

```
System.out.println(mp.get("priya"));//1
System.out.println(mp.get("pra"));//null
System.out.println(mp.getOrDefault("pra",0));//0
System.out.println(mp.getOrDefault("priya",1 ));//0
System.out.println(mp.containsKey("priya"));//true
//Map<String, Integer> map;
//graph
Map<Integer,List<Integer>> adj=new HashMap<>();
// if(adj.get(1)==null) adj.put(1, new ArrayList<>());
// adj.get(1).add(2);
//
adj.computeIfAbsent(1,f-> new ArrayList<>()).add(2);
//
Set<Map.Entry<String,Integer>> entrySet= mp.entrySet();
for(Map.Entry<String,Integer> entry : entrySet)
System.out.println(entry.getKey()+","+entry.getValue());
//priya,1
// pri,3
// riya,2
//
Set<String> keySets=mp.keySet();
for(String key: keySets) System.out.println(key+","+mp.get(key));
//priya,1
// pri,3
// riya,2
    }
}
```

SortedMap: keys are sorted

Navigable Map

TreeMap

```
public class CollectionTest {
    public static void main(String[] args) {
//creating TreeMap
NavigableMap<Integer, String> tmap= new TreeMap<>();
tmap.put(1,"priya");
tmap.put(2,"rima");
tmap.put(3,"mondu");
Set<Entry<Integer, String>> entrySet= tmap.entrySet();
```

```
for(Entry<Integer, String> entry : entrySet)
System.out.println(entry.getKey()+","+entry.getValue());
System.out.println(tmap.ceilingKey(2));
    }
}
```

Extra Functions

```
package Riddhi_Collection;
import java.util.*;
class Sorting
{
    public static void main(String[] args) {
        Integer arr[]=new Integer[]{7,8,9,1,3,2,-9,-6,-10};
        Arrays.sort(arr);//Dual-Pivot Quicksort //array type int[] would
also work above
        //O(N log N ) is time complexity
        // System.out.println(arr);//this prints the toString in Object
class as we print arr object reference
        for(int ele:arr)
            System.out.print(ele+" ");
        // Arrays.sort(arr,Collections.reverseOrder());
        // we can use comparator above ,but type of arr should be
Integer[] and not int[]
        // -10 -9 -6 1 2 3 7 8 9  is output
        System.out.println();
        Integer arr1[]=new Integer[]{7,8,9,1,3,2,-9,-6,-10};
        List<Integer>list=Arrays.asList(arr1);//Arrays.asList(5,4,3,2,1)
would work too
        // arr1 of type int[] won't work here
        // Collections.sort(list);//this is sorting in ascending order
//O(N log N) is time complexity
        Collections.sort(list,Collections.reverseOrder());//2nd argument
is the comparator
        //list cannot be of type int[]
        // 1st argument is of type List
        // for(int ele:list)
        //     System.out.print(ele+" ");
        // 9 8 7 3 2 1 -6 -9 -10 is output
        //above for loop works but below is better way of printing
```

```
        System.out.println(list);
        // [9, 8, 7, 3, 2, 1, -6, -9, -10] is output
        System.out.println();

    }
}
```

THANK YOU ,

Made by -Priya Pramanick

Linkedin- https://www.linkedin.com/in/priya-pramanick-06b41121b/

Twitter - https://twitter.com/priya_nick_17

GitHub-https://github.com/priya-172