**Comparable Interface in TreeSet Example**

import java.util.\*;  
  
class TreeSetCustomSortingDemo  
{  
    public static void main(String arg[])  
    {  
        TreeSet<Student> students = new TreeSet<Student>(); // LINE B  
        students.add(new Student("Sreeram", 10, 'A'));  
        students.add(new Student("Karthik", 12, 'A'));  
        students.add(new Student("Ram", 14, 'B'));  
        students.add(new Student("Yeshwanthi", 5, 'C'));  
        students.add(new Student("Bhavya", 6, 'D'));  
        System.out.println(students); // LINE C    }  
}  
  
class Student implements Comparable<Student>  
{  
    String name;  
    int rollNumber;  
    char section;  
      
    public Student(String name, int rollNumber, char section)  
    {  
        this.name = name;  
        this.rollNumber = rollNumber;  
        this.section = section;  
    }  
      
    public String toString()  
    {  
        return name + " " + rollNumber + " " + section;  
    }  
      
    public int compareTo(Student o)  
    {  
        return rollNumber - o.rollNumber;  
    }  
}

In the above program we have demonstrated how to change the natural ordering of the TreeSet depending on the requirements. Firstly we have created a Student class which implements Comparable interface with the fields name, rollNumber, section and implemented toString and compareTo method of Comparable interface. At LINE B in the other class we have created a TreeSet and added Student objects. Now Set will consider the compareTo method in the Student class and sorts the objects accordingly.

**Sort using section**

**Iterator**

The Iterator helps you to move through all the elements in a collection.

The methods used by Iterator are:

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean hasNext() | Returns true if the collection has next element, else it returns false. |
| E Next() | Returns the next element. If there is no next element, it throws anexception. |
| void remove() | Removes the current element to which the iterator is pointing. |

import java.util.\*;  
  
class IteratorDemo  
{  
    public static void main(String arg[])  
    {  
        ArrayList<Integer> numbers = new ArrayList<Integer>();  
        numbers.add(10);  
        numbers.add(20);  
        numbers.add(30);  
        numbers.add(40);  
        numbers.add(50);  
        Iterator<Integer> itr = numbers.iterator();  
        while (itr.hasNext())   
        {  
            int number = itr.next();  
            System.out.print(number + " ");  
            if (number == 30)  
                itr.remove();  
        }  
        System.out.println("\n..................");  
        itr = numbers.iterator();//LINE A  
        while (itr.hasNext())   
        {  
            int number = itr.next();  
            System.out.print(number + " ");  
        }          
    }  
}

**ListIterator**

ListIterator is used to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

ListIterator is available to only those collections that implement the **Java List Interface** . A listIterator has no current element, it's cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().  
  
Advantages of using listIterator over iterator:

* With iterator you can move only forward, but with listIterator you can also move reverse while reading the elements.
* With listIterator you can obtain the index at any point while traversing, which is not possible with iterator.
* With iterator you can only check whether the next element is available or not, but with listIterator you can check the previous and next elements.
* With listIterator you can add or modify an element at any point while traversing, which is not possible with iterator.
* import java.util.\*;  
    
  class ListIteratorDemo  
  {  
      public static void main(String arg[])  
      {  
          ArrayList ar = new ArrayList();  
          ar.add("Black");  
          ar.add("Red");  
          ar.add("Blue");  
          ListIterator litr = ar.listIterator();  
          while (litr.hasNext()) // In forward direction  
          {  
              System.out.print(litr.next() + " ");  
          }  
          System.out.println();  
          while (litr.hasPrevious()) // In reverse direction  
          {  
              System.out.print(litr.previous() + " ");  
          }  
          System.out.println();  
          litr = ar.listIterator(2); // LINE A - Set iterator at specified index  
          System.out.println(litr.previousIndex() + " " + litr.nextIndex()); // Indices  
          litr.add("Orange"); // LINE B  
          System.out.println("After adding Orange : " + litr.previous());  
          litr.remove(); // LINE C  
          System.out.println("After removing : " + litr.previous());  
          litr.set("Yellow"); // LINE D  
          System.out.println("After setting : " + litr.next());      
      }  
  }

**Java Map**

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

**Java Map Hierarchy**

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:

Java Map Hierarchy

A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value

**HashMap**

import java.util.HashMap;  
import java.util.Map;  
  
class MapTest  
{  
    public static void main(String arg[])  
    {  
        Map<String, String> m = new HashMap<String, String>();  
        Map<String, String> m1 = new HashMap<String, String>();  
        System.out.println("Map objects equal : " + m.equals(m1)); // Equals method  
          
        m.put("8", "Prakash"); // Put method  
        m.put("31", "Shabaz");  
        m.put("12", "Raj ");  
        m.put("14", "Praveen");  
        m.put("5", "Gopi");  
          
        System.out.println("Map Elements : " + m); // Display map elements  
        System.out.println("Size of Map : " + m.size());  
        System.out.println("Key contains : " + m.containsKey("31")); // LINE A  
        System.out.println("Value contains : " + m.containsValue("Raj")); // LINE B  
        System.out.println("Gets the value of key : " + m.get("14")); // LINE C  
        System.out.println("Hash code for map : " + m.hashCode());  
        System.out.println("Is map empty : " + m.isEmpty());  
        System.out.println("Removes the key value : " + m.remove("12")); // LINE D  
        System.out.println("Key set : " + m.keySet()); // LINE E  
        System.out.println("Collection values : " + m.values()); // LINE F  
          
        m1.putAll(m); // LINE G  
        m.clear();  
        System.out.println("Map after clear : " + m);  
        System.out.println("All data of m is put into m1 map : " + m1);  
        System.out.println("Entry set : " + m1.entrySet());      
    }  
}

**Sorted Map**

The SortedMap interface extends Map. It ensures that the entries are maintained in ascending key order.

import java.util.Iterator;  
import java.util.Map;  
import java.util.Set;  
import java.util.TreeMap;  
  
class SortedMapTest  
{  
    public static void main(String arg[])  
    {  
        TreeMap<String, Double> tm = new TreeMap<String, Double>();  
        tm.put("Santosh", new Double(3020.55));  
        tm.put("Ram", new Double(2550.22));  
        tm.put("Nishan", new Double(2060.66));  
        tm.put("Amar", new Double(1890.88));  
        tm.put("Om", new Double(1650.11));  
        System.out.println("Map after initialization : " + tm);  
          
        Set set = tm.entrySet(); // Setting entry set  
        Iterator i = set.iterator(); // Iterating to set  
        while (i.hasNext()) {  
            // Assigning iterator to map entry  
            Map.Entry m = (Map.Entry) i.next();  
            if (m.getKey().equals("Ram")) {  
                m.setValue(3550.33); // Set value for Ram key  
            }  
            // Getting key and value from map entry  
            System.out.println(m.getKey() + " : " + m.getValue());  
        }  
        System.out.println("Map after changing Ram value : " + tm);  
          
        System.out.println("First key : " + tm.firstKey());  
        System.out.println("Last key : " + tm.lastKey());  
        System.out.println("Keys set : " + tm.keySet());  
        System.out.println("Values set : " + tm.values());  
        System.out.println("Head map : " + tm.headMap("Om"));  
        System.out.println("Sub map : " + tm.subMap("Nishan", "Santosh"));  
        System.out.println("Tail map : " + tm.tailMap("Om"));  
      
    }  
}

**Map.Entry Interface**

The Map.Entry interface enables you to work with a map entry.

The entrySet() method declared by the Map returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

import java.util.\*;  
  
class MapEntryTest  
{  
    public static void main(String arg[])  
    {  
        HashMap<String, Double> hm = new HashMap<String, Double>();  
        hm.put("Santosh", new Double(3020.55));  
        hm.put("Ram", new Double(2550.22));  
        hm.put("Nishan", new Double(2060.66));  
        hm.put("Amar", new Double(1890.88));  
        hm.put("Om", new Double(1650.11));  
        System.out.println("After initializing : " + hm);  
          
        Set set = hm.entrySet(); // Setting entry set  
        Iterator i = set.iterator(); // Iterating to set  
        while (i.hasNext()) {  
            // Assigning iterator to map entry  
            Map.Entry m = (Map.Entry) i.next();  
            if (m.getKey().equals("Ram")) {  
                m.setValue(3550.33); // Set value for Ram key  
            }  
            // Getting key and value from map entry  
            System.out.println(m.getKey() + " : " + m.getValue());  
        }  
        System.out.println("After changing : " + hm);      
    }  
}

**Tree Map**

TreeMap is a Red-Black Tree based Map implementation. The TreeMap is sorted based on the natural ordering of its keys or by a Comparator provided in the constructor.

* The key-value value pairs in a TreeMap are sorted.
* TreeMap stores only unique key values. When it encounters a key value that is already present in the map it updates the new entry (on the previous entry with new value for the specified key).
* Duplicate values can be present in a TreeMap.
* import java.util.TreeMap;  
    
  class TreeMapTest  
  {  
      public static void main(String arg[])  
      {  
          TreeMap<Integer, String> studentmarks = new TreeMap<>(); // LINE A  
          studentmarks.put(4, "Nyamath");  
          studentmarks.put(2, "Pramod");  
          studentmarks.put(5, "Amar");  
          studentmarks.put(1, "Santosh");  
          studentmarks.put(3, "Soumya"); // LINE B  
          //Automatically sorts in ascending key order   
          System.out.println("Map sorted in ascending key order : " + studentmarks); // LINE B  
          // poll.FirstEntry returns the first entry in the  map and then it removes from map  
          System.out.println("First entry in the map is : " + studentmarks.pollFirstEntry()); // LINE C  
          System.out.println("Map after using pollFirstEntry : " + studentmarks);      
      }  
  }

**Weak HashMap**

WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbagecollected when its key is no longer referenced outside of the WeakHashMap.

import java.util.\*;  
  
class WeakHashMapTest  
{  
        public static void main(String args[])  
        {      
          Map weakmap=  (Map) new WeakHashMap();  
          String one=new String("one");  
          String two=new String("two");  
          weakmap.put(one, "Lahari");  
          weakmap.put(two, "Latha");  
          System.gc();  
          System.out.println("Before: "+weakmap.get("one")+" "+weakmap.get("two"));  
          one=null;  
          two=null;  
           System.gc();  
          System.out.println("After: "+weakmap.get("one")+" "+weakmap.get("two"));     
        }  
}

**LinkedHash Map**

The Class LinkedHashMap is an extension of Java HashMap Implementation with specific feature of retaining the insertion order in the order, in which they were inserted. Also if one inserts the key again into the LinkedHashMap the original orders is retained.

This allows insertion-order iteration over the Java Map Interfaces - HashMap, TreeMap, LinkedHashMap. That is, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted. You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

import java.util.\*;  
  
class LinkedHashMapTest  
{  
    public static void main(String arg[])  
    {  
        LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();  
        lhm.put("Ramesh", "Intermediate");  
        lhm.put("Shiva", "B-Tech");  
        lhm.put("Santosh", "B-Com");  
        lhm.put("Asha", "Msc");  
        lhm.put("Raghu", "M-Tech");  
          
        Set set = lhm.entrySet();  
        Iterator i = set.iterator();  
        while (i.hasNext()) {  
            Map.Entry me = (Map.Entry) i.next();  
            System.out.println(me.getKey() + " : " + me.getValue());  
        }  
                  
        System.out.println("The Key Contains : " + lhm.containsKey("Shiva"));  
        System.out.println("The value to the corresponding to key : " + lhm.get("Asha"));      
    }  
}

**Identity HashMap**

IdentityHashMap is similar to HashMap except for the following differences:

1) In case of HashMap, JVM will always use equals() method to identify duplicate keys.

2) In case of IdentityHashMap, as its name suggests, JVM will always use == operator to identify duplicate keys.

IdentityHashMap was added in Java 1.4. It implements Map interface. But it doesn’t use equals() and hashCode() methods for comparing objects unlike other implementations of Map (e.g., HashMap).

Which is faster HashMap or IdentityHashMap?

Since IdentityHashMap uses == operator for comparing objects, it is faster compared to HashMap.

When to use IdentityHashMap?

It is suitable where we need reference equality check instead of logical equality.

NOTE:

Any implementation of Map interface is supposed to use equals() to compare objects, but IdentityHashMap doesn’t do that. **So, it violates Map’s general contract.** Hence it is easy to see that it is a special kind of Map which is rarely used.

import java.util.\*;  
  
class IdentityHashMapDemo  
{  
    IdentityHashMap hm = new IdentityHashMap();  
    Integer i1 = new Integer(10);  
    Integer i2 = new Integer(10);  
    hm.put(i1, "Sachin");  
    hm.put(i2, "Dravid");  
    System.out.println(hm);  
}

**Java Comparators**

Both Java TreeSet - TreeSet Examples in Java and TreeMap In Java - java.util.TreeMap store elements in sorted order. However, it is the Comparator that defines precisely what sorted order means.

The Comparator interface defines two methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| int compare(Object obj1, Object obj2) | Returns zero if the objects obj1 and obj2 are equal. Returns a positive value if obj1 is greater than obj2 Otherwise a negative value is returned. |
| boolean equals(Object obj) | Returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false. |

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
  
class ComparatorTest  
{  
    public static void main(String arg[])  
    {  
        List<Student> list = new ArrayList<Student>(); // List of Student Type  
        list.add(new Student("Nishan", 538, 'A')); // Adding Elements to list  
        list.add(new Student("Ram", 513, 'A'));  
        list.add(new Student("Sanath", 583, 'B'));  
        list.add(new Student("Amar", 504, 'A'));  
        list.add(new Student("Shiva", 506, 'A'));  
        list.add(new Student("Bobby", 589, 'B'));  
        Collections.sort(list, new Comparator<Student>() { // Comparator in Collections sort  
          
                    public int compare(Student a1, Student a2) { // Compare method  
                        return a1.name.compareTo(a2.name); // LINE A  
                    }  
                });  
        for (int i = 0; i < list.size(); i++) {  
            Student a = list.get(i); // Assigning list element to Student object  
            System.out.println(a.name + " " + a.rollno + " " + a.section);  
        }  
      
    }  
}  
  
class Student {  
  
    String name;  
    int rollno;  
    char section;  
  
    public Student(String name, int rollno, char section) {  
        this.name = name;  
        this.rollno = rollno;  
        this.section = section;  
    }  
}