SortedSet:

* It is sub interface of set
* It ensures that the elements are stored in the sorted order.

Key Features:

Maintains elements in ascending order by default.

Does not allow duplicate elements.

TreeSet Class:

(Thread-unsafe)

[TreeSet (Java SE 21 & JDK 21)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html)

package sortedSetExamples;  
  
import java.util.TreeSet;  
  
public class TreeSetExample {  
 public static void main(String[] args) {  
 TreeSet<Integer> treeSet = new TreeSet<>();  
  
 //Adding elements  
 treeSet.add(60);  
 treeSet.add(40);  
 treeSet.add(30);  
 treeSet.add(10);  
  
 System.*out*.println("TreeSet: "+ treeSet);  
  
 //1. get first element(smallest)  
 System.*out*.println("First Element: "+ treeSet.first());  
  
 //2. get last element(largest)  
 System.*out*.println("Last Element: "+ treeSet.last());  
  
 //3. get elements less than 30  
 System.*out*.println("Elements(less than 30): "+treeSet.headSet(30));  
  
 //4. to get elements greater than or equal to 30  
 System.*out*.println("Elements(greater than 30 or equal): "+ treeSet.tailSet(30));  
  
 //5. get subset of elements between 10 and 40  
 System.*out*.println("Subset (10,40): "+ treeSet.subSet(10,40));  
 }  
}

Queue:

* It follows the FIFO(First In First Out) principle

PriorityQueue:

package queueExample;  
  
import java.util.PriorityQueue;  
  
public class PriorityQueueExample {  
 public static void main(String[] args) {  
 PriorityQueue<Integer> pq = new PriorityQueue<>();  
  
 //Add elements in pq  
 pq.add(40);  
 pq.add(30);  
 pq.add(10);  
 pq.add(20);  
  
 //Printing the pq  
 System.*out*.println("priorityQueue: "+ pq);  
 // for integer objects, smallest number will have the highest priority  
  
 //Accessing the highest priority element(peek at the top)  
 System.*out*.println("Highest Priority Element : "+ pq.peek());  
  
 // Removing the highest priority element  
 System.*out*.println("Poll: "+ pq.poll());  
  
 System.*out*.println("priorityQueue: "+ pq);  
  
 System.*out*.println("Size: "+ pq.size());  
 }  
}

o/p:

priorityQueue: [10, 20, 30, 40]

Highest Priority Element : 10

Poll: 10

priorityQueue: [20, 40, 30]

Size: 3

MAP Interface:

* A map is a collection in java used for storing key-value pairs.
* Each key must be unique but we can store duplicate values.
* It is a part of java.util
* Common Implementations: HashTable, HashMap, LinkedHashMap, TreeMap.

Why?

* Efficient retrieval of data using key.
* Suitable for scenarios where a relation exist between key, value pairs.

Methods of Map:

Put(K key, V value): Adds a key-value pairs

Get(Object Key): retrieve the value associated with the key

Remove(Object key): Removes the key-value pair

containsKey(Object Key): Checks If a key is present

containsValue(Object Value): Check if a value is present

values(): Returns a collection of values.

[Map (Java SE 21 & JDK 21)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Map.html)

1.HashTable:

Features:

* Synchronized and thread safe
* Suitable for multi-threaded environment.
* Does not allow null keys or values

package mapExample;  
  
import java.util.Hashtable;  
  
public class HashTableExample {  
 public static void main(String[] args) {  
 //create hashtable  
 Hashtable<Integer, String> table = new Hashtable<>();  
  
 //to add elements(key-value pairs)  
 table.put(1,"JAVA");  
 table.put(2, "Python");  
 table.put(3, "C#");  
 System.*out*.println("Before removal : "+ table);  
 //to retrieve the value by key  
 System.*out*.println("Value of 3rd key: "+table.get(3));  
  
 //to check if key exists or not  
 System.*out*.println("Contains key 4 ? : "+table.containsKey(4));  
  
 //to remove the k-v pair  
 table.remove(3);  
 System.*out*.println("After removal : "+ table);  
  
 }  
}

o/p:

Before removal : {3=C#, 2=Python, 1=JAVA}

Value of 3rd key: C#

Contains key 4 ? : false

After removal : {2=Python, 1=JAVA}

HashMap:

Features:

* Not Synchronized(Not thread-safe)
* Allows one null key and multiple null values
* Faster than hashtable in single threaded environments.
* It doesn’t maintains the order of elements.
* package mapExample;  
    
  import java.util.HashMap;  
    
  public class HashMapExample {  
   public static void main(String[] args) {  
   //creating hashmap  
   HashMap<String, Integer> map = new HashMap<>();  
    
   //to add key value pairs  
   map.put("Apple", 100);  
   map.put("Banana", 50);  
   map.put("Cherry", 75);  
    
   //method to retrieve value by key  
   System.*out*.println("Price of Apple: "+ map.get("Apple"));  
    
   // method to check if key exist or not  
   System.*out*.println("Does cart contain banana? "+ map.containsKey("Banana"));  
    
   //containsValue()  
   //remove()  
    
   //to get all values  
   System.*out*.println("Values: "+ map.values());  
    
    
   // to retrieve all key-value pairs  
   System.*out*.println("All Values: "+ map.entrySet());  
   }  
  }

o/p

Price of Apple: 100

Does cart contain banana? true

Values: [100, 75, 50]

All Values: [Apple=100, Cherry=75, Banana=50]

LinkedHashMap:

Features:

* Maintains the insertion order
* Not synchronized(Not thread-safe)
* Allows one null key and multiple null values
* Slower than the hashmap due to maintaining the insertion order.

package mapExample;  
  
import java.util.LinkedHashMap;  
  
public class LinkedHashMapExample {  
 public static void main(String[] args) {  
 //Creating the LHM  
 LinkedHashMap<String, String> linkedMap = new LinkedHashMap<>();  
  
 //adding key-values  
 linkedMap.put("Krishna", "CEO");  
 linkedMap.put("Govind", "Developer");  
 linkedMap.put("Gopal", "Tester");  
  
 //Retrieve value by key  
 System.*out*.println("Role of Gopal: "+ linkedMap.get("Gopal"));  
  
 //containsKey()  
 //containsValue()  
 //remove  
 //keySet()  
 // Returns a Set view of the keys contained in this map.  
  
 System.*out*.println("Values: "+ linkedMap.values());  
  
 //entrySet()  
 }  
}

o/p:

Role of Gopal: Tester

Values: [CEO, Developer, Tester]

When to use:

Hashtable: If thread safety is required.

HashMap: for faster operations when thread safety is not necessary

LinkedHashMap: if the order of insertion is necessary