Day12: Map, Arrays and Collections classes, Iterator, Generics, Queue, Priority Queue

Arrays and Collections classes:

Arrays class:

The **Arrays** class in **java.util package** is a utility class that provides various **static** methods which is helpful to forms various utility methods on the normal array in java.

for example, printing array's elements, sorting an array, searching any elements from an array etc.

Some of the important methods of **java.util.Arrays** class are :

Method	**Description**
public static void sort(originalarray)	Sorts the complete array in ascending order.
public static String toString(originalarray)	It returns a string representation of the contents of this array as comma separated.
public **static <T> List<T> asList(T... a)**	This method returns a fixed-size list backed by the specified array.
public **static int binarySearch(**originalarray, searchkey**);**	it search a element from the specified sorted array, if element founds then returns the index of the element, otherwise it returns negative number. (element must be sorted)
public **static boolean equals(array1, array2)**	to compare 2 array's having the same element

Eample1 : sorting and printing array elements

```
int[] arr = {10,2,5,12,15,16,7,6};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));
```

Example2: creating List of String in easier way :

```
List<String> list= Arrays.asList("one","two","three","four","five");

System.out.println(list);


output:
[one, two, three, four, five]
```

.

Collections class:

This class belongs to **java.util** package, it is also a utility class that provides several **static** utility methods to perform various utility operations on the collection of objects like searching, sorting, reversing a collection of objects, etc.

some of the methods of Collections class:

Method	Description
public static void reverse(List list)	Reverse the elements of the specified List
public static boolean replaceAll(List list, Object old, Object new);	Replaces all occurrences of one specified value in a list with another.
public static void sort(List list)	Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
public static void sort(List list, Comparator comp)	Sorts the specified list according to the order induced by the specified comparator.
public static List synchronizedList(List list);	Returns a synchronized (thread-safe) list backed by the specified list.

public static int frequency(Collection c, Object o);	Returns the number of elements in the specified collection equal to the specified object.
public static void copy(List dest, List src);	Copies all of the elements from one list into another.
public static boolean addAll(Collection c, Object... elements);	Adds all of the specified elements to the specified collection.
public static Set synchronizedSet(Set s);	Returns a synchronized (thread-safe) set backed by the specified set.

Example: Sorting a List of Integer in natural sorting order:

```
        List<Integer> list= Arrays.asList(10,5,8,2,12,15,14,1,9);

        System.out.println("Before sorting List "+ list);

        Collections.sort(list);


        System.out.println("After sorting List "+ list);
```

Example2: Sorting a List of Integer in descending order:

```
      List<Integer> list= Arrays.asList(10,5,8,2,12,15,14,1,9);

        System.out.println("Before sorting List "+ list);

        Collections.sort(list);
        Collections.reverse(list);

        System.out.println(list);
```

I Problem:

Sorting a List of Student using Comparator (according to the marks)

```
//Student.java
public class Student  {
```

```
    private int roll;
    private String name;
    private int mark;

    public Student() {
    }


    public Student(int roll, String name, int mark) {
        this.roll = roll;
        this.name = name;
        this.mark = mark;
    }

    public int getRoll() {
        return roll;
    }

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

    public String getName() {
        return name;
    }

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

    public int getMark() {
        return mark;
    }

    public void setMark(int mark) {
        this.mark = mark;
    }
}


//StudentMarksComp.java
import java.util.Comparator;
class StudentMarksComp implements Comparator<Student> {


    @Override
    public int compare(Student s1, Student s2) {

        if(s1.getMark() > s2.getMark())
            return +1;
        else if(s1.getMark() < s2.getMark())
            return -1;
        else
            return 0;
```

```
    }

}

//Main.java

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
class Main {

    public static void main(String[] args) {

        List<Student> students = new ArrayList<>();

        students.add(new Student(10,"name1",850));
        students.add(new Student(12,"name2",450));
        students.add(new Student(14,"name3",950));
        students.add(new Student(15,"name4",550));
        students.add(new Student(16,"name5",650));


        Collections.sort(students,new StudentMarksComp());

        for(Student student: students){
            System.out.println("Roll :"+student.getRoll());
            System.out.println("Name :"+student.getName());
            System.out.println("Marks :"+student.getMark());
            System.out.println("==============================");
        }
    }
}
```

Map In Java:

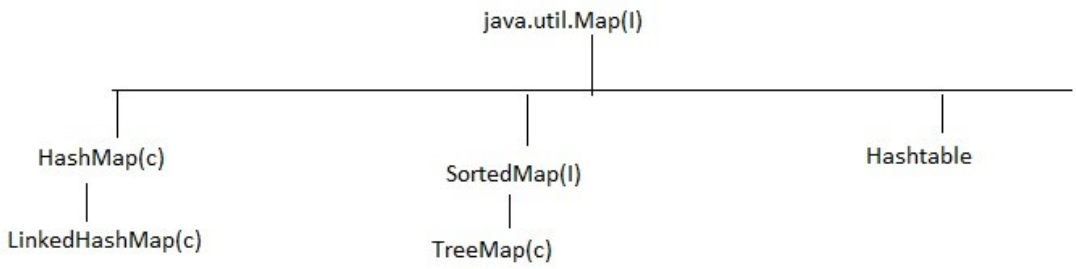
Using Map also we can group multiple objects in the form of key-value pair. here each key-value pair is known as an element.

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

Example:

- A map of error codes and their descriptions.

- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).



Map Interface:

- It is not a child interface of Collection interface.
- This interface belongs to **java.util** package.
- Map able to arrange all the elements in the form of key-value pairs.
- In Map, both keys and values are objects.

Some of the Important Methods of Map:

Method	Description
public V put(K key, V value)	Associates the specified value with the specified key in this map. It is used to insert an entry in the map.
public V get(Object key)	This method returns the object that contains the

	value associated with the key.
public V remove(Object key)	It is used to delete an entry for the specified key.
public void clear()	It is used to reset the map.
public int size()	This method returns the number of entries in the map.
public boolean containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
public boolean containsValue(Object value);	Returns true if this map maps one or more keys to the specified value.
public Set<K> keySet()	Returns a `Set` view of the keys contained in this map.
public Collection<V> values()	Returns a `Collection` view of the values contained in this map.
public Set<Map.Entry<K,V>> entrySet()	Returns a `Set` view of the mappings contained in this map(set of Entry).
public void putAll(Map map)	Copies all of the mappings from the specified map to this map

Map.Entry Interface:

Entry is the inner interface defined inside the Map interface. So we will be accessed it by Map.Entry name. It returns a collection-view of the map.

Without existing Map object there is no-chance of existing entry object(because of that Entry interface is defined inside Map interface).

Each key value pair is called as one entry ,hence map is considered as a group of entries.

Methods of Map.Entry interface:

- **public K getKey()**; It is used to obtain a key.
- **public V getValue()**; It is used to obtain a value.

- **public V setValue(V value)**; It is used to replace the value corresponding to this entry with the specified value.

Note: Since Entry is an inner interface defined inside the Map interface, we define the variable of this Entry interface with the help of its outer interface Map.

Example:

```
Map.Entry me;
```

HashMap class:

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the value of the corresponding key.

Elements will be inserted based on hash code of the "key".so it does not follow the sequence.

We can add only one null value as a key, but any number of null can be added as value.

Example:

```
import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

        HashMap<Integer,String> hm = new HashMap<>();
        System.out.println(hm);//{}
        hm.put(1,"one");
        hm.put(2,"two");
        hm.put(3,"three");

        System.out.println(hm);

        hm.put(null,"four"); //one null is allowd as a key
        hm.put(null,"FOUR"); //four will be replaced with FOUR
        hm.put(2,"TWO"); // two will be replaced with TWO
```

```
        hm.put(5,null);
        hm.put(6,null); //as a value any number of null be allowed

        System.out.println(hm.size());
        System.out.println(hm);

    }
}

Output:
{}
{1=one, 2=two, 3=three}
6
{null=FOUR, 1=one, 2=TWO, 3=three, 5=null, 6=null}
```

Example 2: Iterating over Map:

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class Main {

    public static void main(String[] args) {

        Map<Integer,String> hm = new HashMap<>();
        hm.put(1,"one");
        hm.put(2,"two");
        hm.put(3,"three");
        hm.put(4,"four");
        hm.put(5,"five");

        System.out.println("Getting all the keys");
        Set<Integer> keys = hm.keySet();
        for(Integer x:keys){
            System.out.println(x);
        }

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

        System.out.println("Getting all the values");
        Collection<String> values= hm.values();
        for(String value:values){
            System.out.println(value);
        }
        System.out.println("=================");
```

```
        System.out.println("Getting both key and values");

        Set<Map.Entry<Integer,String>>  keyValue= hm.entrySet();

        for(Map.Entry<Integer,String> me: keyValue){

            System.out.println(me.getKey()+"======"+me.getValue());
        }
    }
}
```

We Problem:

Let’s create a HashMap for the State topper students, and print them back.

Example:

```
//Student.java
public class Student  {

    private int roll;
    private String name;
    private int mark;

    public Student() {
    }


    public Student(int roll, String name, int mark) {
        this.roll = roll;
        this.name = name;
        this.mark = mark;
    }

    public int getRoll() {
        return roll;
    }

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

    public String getName() {
        return name;
    }

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

```
    }

    public int getMark() {
        return mark;
    }

    public void setMark(int mark) {
        this.mark = mark;
    }
}


//Main.java
import java.util.*;

public class Main {

    public static void main(String[] args) {

       HashMap<String,Student> hm = new HashMap<>();

       hm.put("Maharastra",new Student(10,"Ganesh",950));
       hm.put("Tamilnadu",new Student(12,"Surya",850));
       hm.put("Telangana",new Student(15,"Venkat",920));
       hm.put("Haryana",new Student(16,"Dinesh",910));
       hm.put("Kerla",new Student(18,"Srinu",880));


       //getting all the state name:
        Set<String> states= hm.keySet();

        for(String state:states){
            System.out.println(state);
        }

        //Getting all the Student as List object.
        Collection<Student> cols =  hm.values();
        List<Student> students = new ArrayList<>(cols);

        //printing all the students from the List
        for(Student student:students){
            System.out.println("Roll :"+student.getRoll());
            System.out.println("Name :"+student.getName());
            System.out.println("Marks :"+student.getMark());
            System.out.println("==========================");
        }

        //printing state wise students details:

        Set<Map.Entry<String,Student>> set = hm.entrySet();

        for(Map.Entry<String,Student> me: set) {

            System.out.println("Toppers Student of State :" + me.getKey());
```

```
            System.out.println("*********************************");

            Student student = me.getValue();

            System.out.println("Roll :" + student.getRoll());
            System.out.println("Name :" + student.getName());
            System.out.println("Marks :" + student.getMark());

        }
    }
}
```

You Problem:

Sort the above HashMap according to the Student Mark(ascending order).

LinkedHashMap class:

It is a child class of LinkedHashMap, and it will preserve the sequence of all the entries.

Example:

```
import java.util.*;
class Main{
 public static void main(String args[]){

  LinkedHashMap<Integer,String> hm=new LinkedHashMap<>();

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");

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

TreeMap class:

TreeMap class implements the **Map** and **SortedSet** interface, here map will be sorted according to the natural ordering of its keys, or by a **Comparator** provided at map creation time, depending on which constructor is used.

If we don’t use Comparator, then the key object must be **Comparable.** Otherwise it will raise a ClassCastException.

The TreeMap in Java does not allow null keys (like HashMap) and thus a **NullPointerException** is thrown. However, multiple null values can be associated with different keys.

Example:

```
public class Main {

    public static void main(String[] args) {

        TreeMap<String,String> tm = new TreeMap<>();
        tm.put("Maharastra","Mumbai");
        tm.put("Telangana","Hydrabad");
        tm.put("Tamilnadu","Chennai");
        tm.put("Karnataka","Bangaluru");
        tm.put("Bihar","Patna");

        System.out.println(tm);
    }
}

Output:
{Bihar=Patna, Karnataka=Bangaluru, Maharastra=Mumbai, Tamilnadu=Chennai, Telangana=Hydrabad}
```

TreeMap class using Comparator:

The Key of the TreeMap should be Comparable, if the key is not Comparable then we need to use the Comparator and specify the that Comparator implemented class object to the TreeMap constructor.

Example: Student as a key and state as a value.

```
//StudentMarksComp.java
import java.util.Comparator;

public class StudentMarksComp implements Comparator<Student> {


    @Override
    public int compare(Student s1, Student s2) {

        if(s1.getMark() > s2.getMark())
            return +1;
        else if(s1.getMark() < s2.getMark())
            return -1;
        else
            return 0;
    }
}

//Main.java
import java.util.*;
public class Main {

    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");

        Set<Map.Entry<Student,String>> set= tm.entrySet();

        for(Map.Entry<Student,String> me: set) {

            System.out.println("Toppers Student of State :" + me.getValue());
            System.out.println("*********************************");

            Student student = me.getKey();

            System.out.println("Roll :" + student.getRoll());
            System.out.println("Name :" + student.getName());
            System.out.println("Marks :" + student.getMark());

        }
    }
}
```

Java Hashtable 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.**

Example:

```
import java.util.*;
class Main{
 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());
  }
 }
}
```

Generics In Java:

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics:

There are mainly 3 advantages of generics.

1. **Type-safety:** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
2. **Type casting is not required:** There is no need to downcast the object.
   ```
   List list = new ArrayList();
   list.add("hello");
   String s = (String) list.get(0);
   ```
   ```
   List<String> list = new ArrayList<String>();
   list.add("hello");
   String s = list.get(0);   // no cast
   ```
3. **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Without Generics, we can add any type of Objects in Collection classes:

Example:

```
import java.util.*;
public class Main {

    public static void main(String[] args) {

      List list = new ArrayList();//non-generic collection

      list.add(10); //Integer object
      list.add("ten"); // String object
      list.add(new Student(10,"Amit",750)); //Student object
    }
}

//Here java compiler won't give any kind of error
```

Now to get the objects from the non-generic Collection classes, we need to downcast the object into the appropriate type.

Example

```
import java.util.*;
public class Main {

    public static void main(String[] args) {

      List list = new ArrayList();

      list.add(10);
      list.add("ten");
      list.add(new Student(10,"Amit",750));

        int x= (Integer)list.get(0);
        System.out.println(x);

       String s= (String)list.get(1);
       System.out.println(s);

        Student student = (Student)list.get(2);
        System.out.println(student);

    }
}
```

Without Generics the ArrayList class sudo code:

```
public class ArrayList{

  public boolean add(Object obj){
    //add the obj to the collection
  }

  public Object get(int index){
    //return the specified index value in the form of Object class
  }

}
```

With Generics the ArrayList class sudo code:

```
public class ArrayList<T>{

  public boolean add(T t){
    //add the t obj to the collection
  }

  public T get(int index){
    //return the specified index value in the form of T class
  }

}
```

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

Creating our own Generic class:

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Example

```
//MyGen.java
class MyGen<T>{

  T obj;

  void add(T obj){
      this.obj=obj;
  }

T get(){
  return obj;
  }

}

//Main.java
public class Main {

    public static void main(String[] args) {

        MyGen<String> gen1 = new MyGen();
        gen1.add("Hello");
```

```
        System.out.println(gen1.get());

        MyGen<Integer> gen2 = new MyGen();
        gen2.add(20);
        System.out.println(gen2.get());

        MyGen<Student> gen3 = new MyGen();
        gen3.add(new Student(10,"Amit",780));
        System.out.println(gen3.get().getName());

    }
}
```

Type Parameters:

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method:

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

Example:

```
public class Main{

   public static <E> void printArray(E[] elements) {
```

```
         for ( E element : elements){
             System.out.println(element );
          }
          System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'M', 'A', 'S', 'A', 'I'};

        System.out.println( "Printing Integer Array" );
        printArray( intArray  );

       System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

Wildcard in Java Generics:

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a **type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype**.

Upper Bounded Wildcards

The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

Syntax:

```
List<? extends Number>
```

here **?** is a wildcard character and Number is a predefined java class belongs to **java.lang** package.

Example:

```
import java.util.*;

abstract class Shape{
  abstract void draw();
}

class Rectangle extends Shape{
  void draw(){
      System.out.println("drawing rectangle");
  }
}
class Circle extends Shape{
  void draw(){
    System.out.println("drawing circle");
  }
}

class Main{

//creating a method that accepts only child class of Shape
  public static void drawShapes(List<? extends Shape> lists){

      for(Shape s:lists){
        s.draw();//calling method of Shape class by child class instance
      }
  }

  public static void main(String args[]){
    List<Rectangle> list1=new ArrayList<Rectangle>();
    list1.add(new Rectangle());

    List<Circle> list2=new ArrayList<Circle>();
    list2.add(new Circle());
    list2.add(new Circle());

    drawShapes(list1);
    drawShapes(list2);
  }
}
```

Lower Bounded Wildcards:

The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

Syntax:

```
List<? super Integer>
```

Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses whereas **List<Integer>** works with the list of type Integer only. So, **List<? super Integer>** is less restrictive than **List<Integer>**

Example: specifying the lower bound wildcards to write the method for List<Integer> and List<Number>.

.

```
import java.util.Arrays;
import java.util.List;

public class Main {

    public static void addNumbers(List<? super Integer> list) {

        for(Object n:list)
        {
              System.out.println(n);
        }



    }
public static void main(String[] args) {

    List<Integer> l1=Arrays.asList(1,2,3);
      System.out.println("displaying the Integer values");
    addNumbers(l1);

    List<Number> l2=Arrays.asList(1.0,2.0,3.0);
      System.out.println("displaying the Number values");
    addNumbers(l2);
}

}
```

What is a priority queue?

As mentioned earlier, a regular queue has a first in first out structure. But in some scenarios we want to process messages in a queue based on their priority and not based on when the message entered the queue.

Priority queues help consumers consume the higher priority messages first followed by the lower priority messages.

Priority queues in Java

Now let's see some actual Java code that will show us how to use priority queues.

Priority queues with natural ordering

Here is some code showing how to create a simple priority queue for strings

```
private static void testStringsNaturalOrdering() {
        Queue<String> testStringsPQ = new PriorityQueue<>();
        testStringsPQ.add("abcd");
        testStringsPQ.add("1234");
        testStringsPQ.add("23bc");
        testStringsPQ.add("zzxx");
        testStringsPQ.add("abxy");

        System.out.println("Strings Stored in Natural Ordering in a Priority Queue\n");
        while (!testStringsPQ.isEmpty()) {
            System.out.println(testStringsPQ.poll());
        }
    }
```

The first line tells us that we are creating a priority queue:

```
Queue<String> testStringsPQ = new PriorityQueue<>();
```

PriorityQueue is available in java.util package.

Priority queue with Java objects

Up to this point, we've seen how we can use strings and integers with priority queues.

In real life applications we would generally be using priority queues with custom Java objects.

Let's first create a class called CustomerOrder which is used to store customer order details:

```
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) {
        return o.orderId > this.orderId ? 1 : -1;
    }

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

    public double getOrderAmount() {
        return orderAmount;
    }
}
```

This is a simple Java class to store customer orders. This class implements **comparable interface,** so that we can decide on what basis this object needs to be ordered in the priority queue.

The ordering is decided by the **compareTo** function in the above code. The line **o.orderId > this.orderId ? 1 : -1** instructs that the orders should be sorted based on descending order of the **orderId** field

Below is the code which creates a priority queue for the CustomerOrder object:

```
CustomerOrder c1 = new CustomerOrder(1, 100.0, "customer1");
CustomerOrder c2 = new CustomerOrder(3, 50.0, "customer3");
CustomerOrder c3 = new CustomerOrder(2, 300.0, "customer2");

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

In the above code three customer orders have been created and added to the priority queue.

When we run this code we get the following output:

```
orderId:3, orderAmount:50.0, customerName:customer3
orderId:2, orderAmount:300.0, customerName:customer2
orderId:1, orderAmount:100.0, customerName:customer1
```

As expected, the result comes in descending order of the **orderId**.

What if we want to prioritize based on orderAmount?

This is again a real life scenario. Let's say that by default the CustomerOrder object is prioritized by the orderId. But then we need a way in which we can prioritize based on orderAmount.

You may immediately think that we can modify the **compareTo** function in the **CustomerOrder c**lass to order based on orderAmount.

But the **CustomerOrder c**lass may be used in multiple places in the application, and it would interfere with the rest of the application if we modify the **compareTo** function directly.

The solution to this is pretty simple: we can create a new custom comparator for the CustomerOrder class and use that along with the priority queue

Below is the code for the custom comparator:

```
 static class CustomerOrderComparator implements Comparator<CustomerOrder> {

        @Override
        public int compare(CustomerOrder o1, CustomerOrder o2)
        {
            return o1.getOrderAmount() < o2.getOrderAmount() ? 1 : -1;
        }
    }
```

This is very similar to the custom integer comparator we saw earlier.

The line `o1.getOrderAmount() < o2.getOrderAmount() ? 1 : -1;` indicates that we need to prioritize based on descending order of **orderAmount**.

Below is the code which creates the priority queue:

```
  CustomerOrder c1 = new CustomerOrder(1, 100.0, "customer1");
        CustomerOrder c2 = new CustomerOrder(3, 50.0, "customer3");
        CustomerOrder c3 = new CustomerOrder(2, 300.0, "customer2");
        Queue<CustomerOrder> customerOrders = new PriorityQueue<>(new CustomerOrderCompara
tor());
        customerOrders.add(c1);
        customerOrders.add(c2);
        customerOrders.add(c3);
        while (!customerOrders.isEmpty()) {
            System.out.println(customerOrders.poll());
        }
```

In the above code we are passing the comparator to the priority queue in the following line of code:

```
Queue<CustomerOrder> customerOrders = new PriorityQueue<>(new CustomerOrderComparator());
```

Below is the result when we run this code:

```
orderId:2, orderAmount:300.0, customerName:customer2
orderId:1, orderAmount:100.0, customerName:customer1
orderId:3, orderAmount:50.0, customerName:customer3
```

We can see that the data comes in descending order of the orderAmount.

References:

https://www.javatpoint.com

https://docs.oracle.com/javase/8/docs/api

https://www.geeksforgeeks.org/

https://www.freecodecamp.org/news/priority-queue-implementation-in-java/