# Programming 2 Lab 6

## Enum Type

An **Enum** type defines a fixed set of constant values; variables of this type must take one of the constant values.

As a class, an **Enum** type can declare fields and methods.

Consider a **Customer** class which has a need to categorize customers according to type of account and their priority level. Let’s define an **Enum** type for **CustomerType**:

**public enum CustomerType {**

**INTERNAL\_ACCOUNT (0.25, 2),**

**EXTERNAL\_ACCOUNT (0.1, 1),**

**EXTERNAL\_NOACCOUNT (0, 3);**

**private final double discount;**

**private final int priorityLevel;**

**CustomerType(double discount, int priorityLevel) {**

**this.discount = discount;**

**this.priorityLevel = priorityLevel;**

**}**

**private double getDiscount() {**

**return this.discount;**

**}**

**private int getPriorityLevel() {**

**return this.priorityLevel;**

**}**

**}**

Note constant values are defined using upper case and that each constant has two fields: a discount and a priority level – these are fixed values.

The **Customer** class can then define a **customerType** field of **CustomerType** and this field can be used in constructors, getters, setters and other methods such as **toString()**.

**public class Customer**

**{**

**private int customerId;**

**private String customerName;**

**private String customerAddress;**

**private String customerEmail;**

**private CustomerType customerType;**

**private static int numberOfCustomers=0;**

**public Customer()**

**{**

**this.customerId = ++numberOfCustomers;**

**this.customerName = null;**

**this.customerAddress = null;**

**this.customerEmail = null;**

**this.customerType = CustomerType.INTERNAL\_ACCOUNT;**

**}**

**public Customer(String customerName, String customerAddress,**

**String customerEmail, CustomerType customerType)**

**{**

**this.customerId = ++numberOfCustomers;**

**this.customerName = customerName;**

**this.customerAddress = customerAddress;**

**this.customerEmail = customerEmail;**

**this.customerType = customerType;**

**}**

…

**public CustomerType getCustomerType()**

**{**

**return customerType;**

**}**

**public void setCustomerType(CustomerType customerType)**

**{**

**this.customerType = customerType;**

**}**

@Override

**public String toString() {**

**return "customer id: " + getCustomerId() + ", " +**

**"customer name: " + getCustomerName() + ", " +**

**"customer address: " + getCustomerAddress() + ", " +**

**"customer email: " + getCustomerEmail() + ", " +**

**"customer type: " + getCustomerType();**

**}**

Note an **Enum** type is **Serializable** if required.

We can then instantiate the **Customer** class in an app as follows:

**Customer c = new Customer("Puyol", "La Pobla de Segur",**

**"charlie@barca.com",**

**CustomerType.EXTERNAL\_ACCOUNT);**

### Exercise

Download the Starter project from **GCULearn**, unzip and open in **NetBeans**. Create an **Enum** type to represent employee roles with salary grade notice period (in months) using the values:

|  |  |  |
| --- | --- | --- |
| Project Leader | A | 6 |
| Analyst | B | 3 |
| Developer | B | 3 |
| Tester | C | 1 |

Modify the **Employee** class to change the type of the employee role to the new **Enum** type.

## HashSet

Recall a **HashSet** class is a concrete implementation of the **Set** interface which cannot hold duplicate elements. We create a **HashSet** object and specify the type of object as follows:

**Set<String> names = new HashSet<>();**

We can add elements to the **HashSet** using the **add()** method:

**names.add("Iniesta");**

**names.add("Messi");**

**names.add("Busquets");**

**names.add("Puyol");**

**names.add("Pique");**

**names.add("Puyol");**

We can display the contents of the collection in three ways:

### toString

Note, the **Enum** type implements the **toString()** method:

**System.out.println(names);**

### for-each

We can use the new version of the **for-each** construct to traverse the collection:

**for(String n : names)**

**System.out.println("name:" + n);**

### iterator

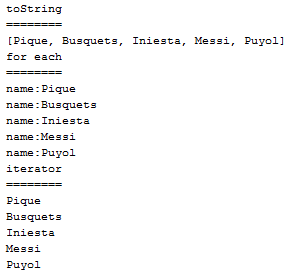
The **Collection** class defines an **iterator()** method which returns an **Iterator** object that can be used to traverse the collection:

**Iterator it = names.iterator();**

**while (it.hasNext())**

**System.out.println(it.next());**

Note the display shows that the second attempt to add “Puyol” fails as it is a duplicate element:



Two other methods are immediately of worth - **contains()** and **remove(** :

**if (names.contains(toBeRemovedElement)) {**

**boolean result = names.remove(toBeRemovedElement);**

**if (result)**

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

**else**

**System.out.println("not present");**

**}**

**else**

**System.out.println("not present");**

### Exercise

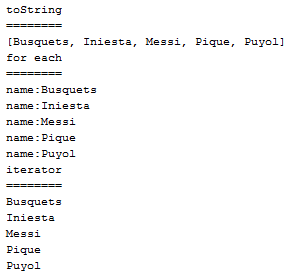
Run the **CollectionsApp** class and ensure you understand the code.

## TreeSet

Recall the **TreeSet** class is a concrete implementation of the **SortedSet** interface where elements are stored in their natural sorted order.

If we change the creation of the **names** attribute object, in the **CustomerStringController** class, to a **TreeSet** then the following display is attained – note no other code requires changing:

**names = new TreeSet<>();**



### Exercise

Make the change in the **CustomerStringCollector** class to a **TreeSet** implementation and run the **CollectionsApp** class and ensure you understand the code.

## HashSet of Objects

Let’s now consider a scenario where the elements of the **Set** collection are not single pieces of data but objects in themselves e.g. a **HashSet** of **Customer** objects.

**Set<Customer> customers = new HashSet<>();**

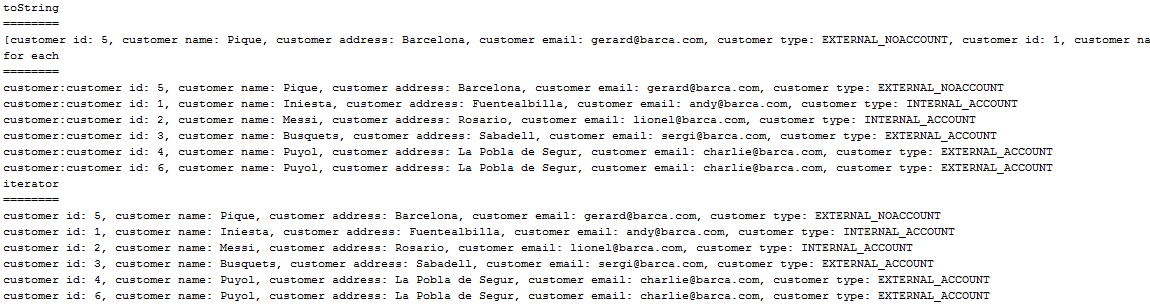
Again an add operation simply involves creating a new **Customer** object and invoking the **add()** method on the **customers** object:

**customers.add(new Customer("Iniesta", "Fuentealbilla",**

**"andy@barca.com",**

**CustomerType.INTERNAL\_ACCOUNT));**

Again we can output the collection in the three ways identified earlier:



Note this time “Puyol” can be added twice as the constructor for the **Customer** class allocates a new value for the **customerId** and therefore the two **Customer** objects are distinct. Ids and their allocation need careful thought when you’re developing business systems.

What about removal? Well again we can use the **contains()** method to determine if an object is contained in the collection and then **remove()** to remove it.

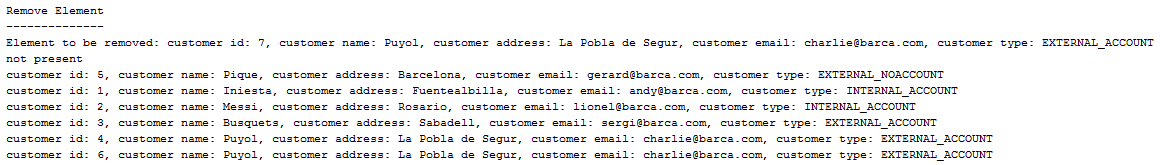
**Customer toBeRemoved =**

**new Customer("Puyol", "La Pobla de Segur", "charlie@barca.com",**

**CustomerType.EXTERNAL\_ACCOUNT);**

**if (customers.contains(toBeRemovedElement)) {**

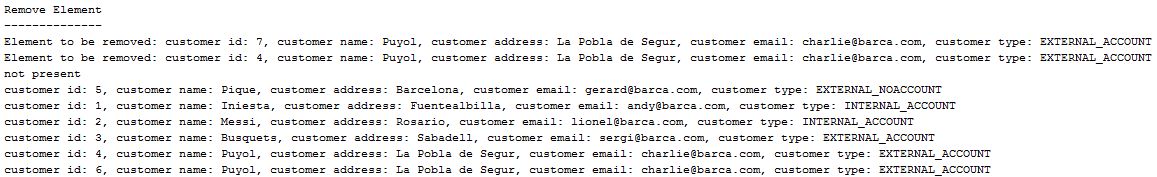
**boolean result = customers.remove(toBeRemovedElement);**



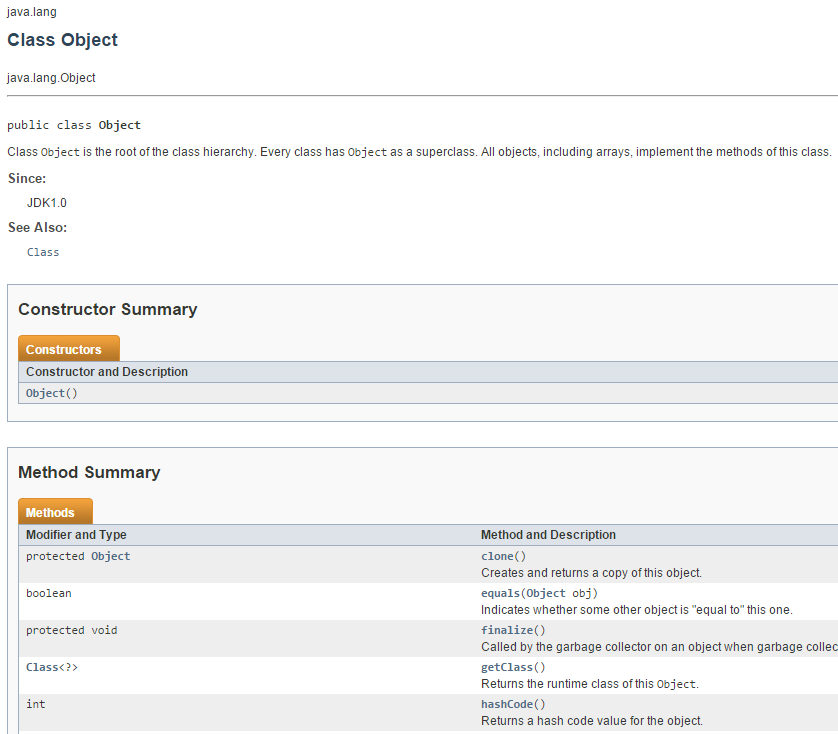
We can see that the removal fails – perhaps due to the fact that the element to be removed has **customerId** 7 while the element to be removed has **customerId** has 4 (or 6).

Let’s use the setter method for **customerId** to change the value of this field for the **toBeRemoved** object:

**toBeRemoved.setCustomerId(4);**



The remove operation still fails as though the two objects contain the same values for fields they are different objects. Recall that all classes in Java ultimately descend from the **Object** class which defines an **equals()** method:



We can override this method in the **Customer** class to change the way in which **Customer** objects are tested for equality:

@Override

**public boolean equals(Object o) {**

**if (o instanceof Customer) {**

**Customer c = (Customer)o;**

**return c.getCustomerId() == getCustomerId() &&**

**c.getCustomerName() == getCustomerName() &&**

**c.getCustomerAddress().equals(getCustomerAddress()) &&**

**c.getCustomerEmail().equals(getCustomerEmail());**

**} else {**

**return false;**

**}**

**}**

We also require a **hashCode()** override, 31 is an arbitrary number:

@Override

**public int hashCode() {**

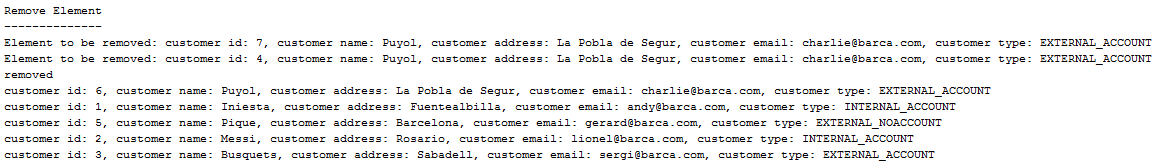
**return getCustomerId() \* 31 + getCustomerName().hashCode() \* 31 +**

**getCustomerAddress().hashCode() \* 31 +**

**getCustomerEmail().hashCode() \* 31;**

**}**

Let’s make the change and try the removal again:



This time we are successful. Note that it only removes the “Puyol” object with the specified **customerId**; however, this is a business rules issue concerning the creation of **Customer** objects rather than a removal issue.

**Customer** objects are equal if the **equals()** method returns true and they have the same hash code. Objects with the same hash code may not be equal.

### Exercise

Uncomment the necessary code in the **Customer** class and modify the **CollectionsApp** **run()** method to create a **CustomerObjectsController** object instead of **CustomerStringController**, run the **CollectionsApp** class and ensure you understand how it works.

### Exercise

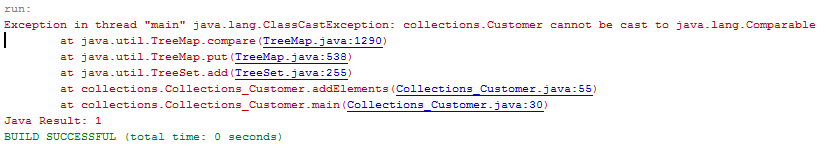
Create an **EmployeeController** class which will implement a **HashSet** of **Employee** objects including adding employees, listing all employees and listing employees with a specified role. Modify the **CollectionsApp** class to create an **EmployeeController** object to test your implemented functionality.

## TreeSet of Objects

What about a sorted set of objects? When we worked with **String**s we could simply change from a **HashSet** to a **TreeSet** without any other changes of code to give us a sorted set; this is because Strings have a natural ordering.

However, there is no natural order associated with the **Customer** object and if we make the change to a **TreeSet** of **Customer** objects and run the application we will produce a runtime error:

**Set<Customer> customers = new TreeSet<>();**



A **TreeSet** needs a **Comparable** implementation in order to add new elements in the correct position. Let’s look at modifying the **Customer** class to handle this:

**public class Customer implements Comparable<Customer>**

@Override

**public int compareTo(Customer compareCustomer) {**

**int custId = ((Customer) compareCustomer).getCustomerId();**

//ascending order

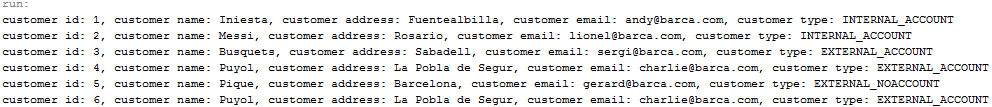
**return this.customerId - custId;**

//descending order

//return customerId - this.custId;

**}**

This gives us a sorted set based on the ‘key’ field **customerId**:



What about sorting on other fields, we need to either change the **compareTo()** method or implement a **Comparator**:

**public static Comparator<Customer>**

**CustomerNameComparator = new Comparator<Customer>()**

**{**

**@Override**

**public int compare(Customer cust1, Customer cust2)**

**{**

**String custName1 = cust1.getCustomerName();**

**String custName2 = cust2.getCustomerName();**

//ascending order

**return custName1.compareTo(custName2);**

//descending order

//return custName2.compareTo(custName1);

**}**

**};**

Note this uses an anonymous class to implement/override the **compare()** method of the **Comparator** interface – recall you cannot instantiate an **Interface** and therefore a class must be instantiated and the interface method executed on that object – an anonymous class means that we do not have to name the class and object that is used to implement the interface and invoke the method.

The controller code will look something like this; we will create a new **TreeSet** object with the elements from the original **TreeSet** but using the **Comparator** for ordering:

**System.out.println("Sort By Name");**

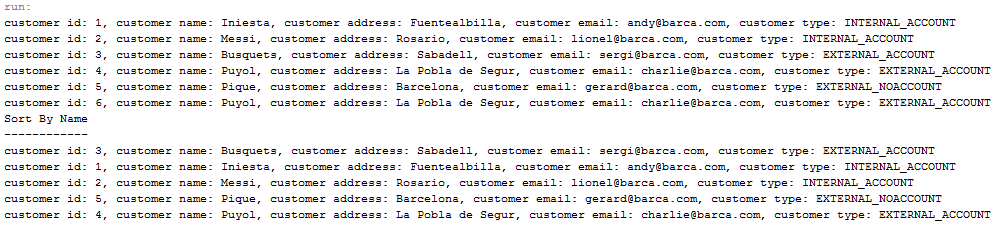
**System.out.println("------------");**

**Set<Customer> newSet = new TreeSet(Customer.CustomerNameComparator);**

**newSet.addAll(customers);**

**displayElements(newSet);**

The output from **displayElements()**:



Note you can create additional **Comparator**s for different types of sorting including complex combinations.

### Exercise

Uncomment the necessary code and check the output is similar to above.

### Exercise

Amend the **EmployeeController** class you created earlier to implement a **TreeSet** of **Employee** objects with the necessary changes to the **Employee** class.

Add a comparator to allow the **Employee**s to be sorted in name order and test.

## List of Objects

An alternative Collection interface is the List interface with concrete implementations: **ArrayList** and **LinkedList**. Let’s look at changing the **Customer** collection from a **Set** to a **List**:

**List<Customer> customers = new ArrayList<>();**



A **List** can hold duplicate elements and elements can be inserted into a specific position; the following code will illustrate this by changing the code to create a **Customer** object and then add it to the collection twice at the start and the end of the collection

**customers.add(new Customer("Iniesta", "Fuentealbilla",**

**"andy@barca.com", CustomerType.INTERNAL\_ACCOUNT));**

**customers.add(new Customer("Messi", "Rosario", "lionel@barca.com",**

**CustomerType.INTERNAL\_ACCOUNT));**

**customers.add(new Customer("Busquets", "Sabadell",**

**"sergi@barca.com", CustomerType.EXTERNAL\_ACCOUNT));**

**customers.add(new Customer("Pique", "Barcelona", "gerard@barca.com",**

**CustomerType.EXTERNAL\_NOACCOUNT));**

**Customer c = new Customer("Puyol", "La Pobla de Segur",**

**"charlie@barca.com", CustomerType.EXTERNAL\_ACCOUNT);**

**customers.add(c);**

**customers.add(0, c);**

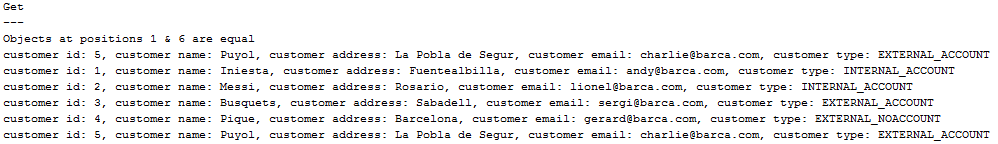
We can use the **get()** method to retrieve the element at the specified index position:

**if (customers.get(0) == customers.get(5))**

**System.out.println("Objects at positions 1 & 6 are equal");**

**displayElements(customers);**

Output:



We can use the **indexOf()** method to determine the index position of the first occurrence of the object and **lastIndexOf()** for the last index position:

**System.out.println("Index Of");**

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

**System.out.println(c);**

**int index = customers.indexOf(c);**

**System.out.println("Index: " + index);**

**System.out.println("Last Index Of");**

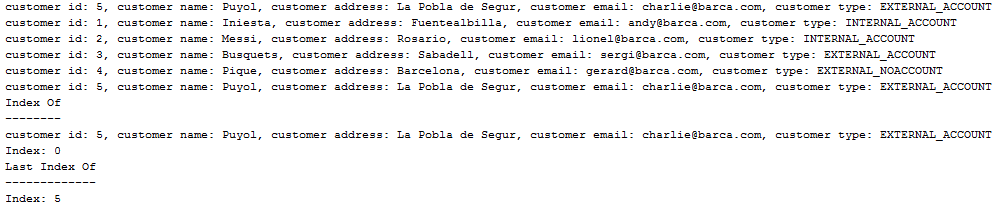
**System.out.println("-------------");**

**System.out.println(c);**

**index = customers.lastIndexOf(c);**

**System.out.println("Index: " + index);**

Output:



We can remove an element by locating its index as above and using the **remove()** method or more commonly using an iterator to traverse the collection and test each element in turn for a match and then removing.

What about sorting? Well a **List** is already sorted in the order in which elements have been inserted, we can use the **sort()** method in conjunction with the **Comparator** to sort by specific fields:

**System.out.println("Sort By Id");**

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

**Collections.sort(customers);**

**displayElements(customers);**

**System.out.println("Sort By Name");**

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

**customers.sort(Customer.CustomerNameComparator);**

**displayElements(customers);**

Output:



### Exercise

Implement the necessary changes to sort by address and test.