RIT | Golisano College of Computing and Information Sciences
School of Information

ISTE-121
Lab 08: Sorting

Overview

Part 1 - Sorting (5 points)

Overview of problem:
From the zip download, open the program SortAddresses. This program sorts a list of addresses by zip code. Sorting only by the one value of zip code, this is considered a one level sort.

Problem to solve:
We need to sort not only by zip code but also alphabetically by state and city. Look at the supplied code, **do not delete** the zip code Comparator, and add a new state/city sort. Have the main print out the sorted state/city list at the indicated place in the code.

This is a two-level sort. Sorting on state code results in a return of -1, 0 or 1 from the compare() method, as it did with the zip code compare() method. However, when a state is found to be the same as another state, the result is 0. In this instance you need to repeat the comparison for city. The discovery of how to do this is left up to you.

Questions to answer:

Place your answers, using the question #, in a text file named Lab08Part1.txt.

1. Explain the class signature of CustomerList.

2. What does "new CustomerList();"accomplish?

Submit your *.java file(s) and the Lab08Part1.txt file to the Lab08 Assignment folder when your program for Part 1 is working correctly.

Part 2 - Table (5 points)

A Table is a GUI control that displays 2-dimensional data in tabular form.

The data underlying (and displayed in) the Table (called the data model) is usually based on the class that represents one row of the table.

Consider the following data:

```
Brock    10203 872564.15
Boyer    10204 985.59
Gardner  10205 15867.30
```

(NOTE: this is just a sample, the file you will be using has many more entries).

We will assume the file is a .dat file and these fields are written using a DataOutputStream (there is no header line).

Each line of this data represents one bank customer, with a name, an ID, and an account balance. In Java we will represent this as a class, Customer, with three attributes (a String called name, an Integer called id, and a Double called balance).

Create the Customer class, with a parameterized constructor (expecting the name, id and balance as parameters) and with an accessor and mutator for each attribute. Get Customer to compile.

We want to display this data in a Table. The way to do this, create an attribute:

```
private TableView<Customer> tblData = new TableView<Customer>();
```

The underlying table model will be an ArrayList built by javafx for us:

```
ObservableList<Customer> data = FXCollections.observableArrayList();
```

Assuming the `data` is in one of the forms described above. The `ObservableList` is an object used to control what is displayed in the `Table` at any time. With more complex controls, that display lots of data, Java uses the model approach, separating the control and how it appears from the model and how the data is structured.

So, let's put this all together. See SimpleTableStarter.java in today's downloads. Also, download BankData.dat to use with this program.

- Declare a class that extends Application and has attributes for the Table, the column headers, and the data model, and other attributes

```
import javafx.application.*;
import javafx.event.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.control.Alert.*;
import javafx.scene.control.cell.*; // needed to do cell formatting
import javafx.scene.layout.*;
import javafx.stage.*;
import javafx.geometry.*;
import javafx.collections.*;        // needed for FXCollections
```

```
import java.io.*;
import java.util.*;

public class SimpleTable1 extends Application {
   // Window Attributes
   private Stage stage;
   private Scene scene;
   private VBox root = new VBox(8);

   // GUI components
   private TableView<Customer> tblBank = new TableView<Customer>(); // The Table
   private ObservableList<Customer> data =
      FXCollections.observableArrayList();

   // The data file
   public static final String BANK_FILE = "BankData.dat";

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

- Setup the window

```
   public void start(Stage _stage) {
      // Setup the window
      stage = _stage;
      stage.setTitle("Simple Table");
//       this.setSize(400, 700);
      stage.setOnCloseRequest(new EventHandler<WindowEvent>() {
         public void handle(WindowEvent evt) { System.exit(0); }
      });
```

- Set up the table columns with a column header and a type
  - The type parameters to TableColumn are the underlying class (Customer) and the type of that particular column (String, Integer or Double)
- Set the preferred height and width of the table
- Add the columns to the Table
- Finally, add the table to the root of the GUI

```
      // Set up table columns
      TableColumn<Customer,String> tcolName =
         new TableColumn<Customer,String>("Name");
      TableColumn<Customer,Integer> tcolId =
         new TableColumn<Customer,Integer>("ID");
      // right align the ID
      tcolId.setStyle( "-fx-alignment: CENTER-RIGHT;");
      TableColumn<Customer1,Double> tcolBalance =
         new TableColumn<Customer,Double>("Balance");
      // right align the balance
      tcolBalance.setStyle( "-fx-alignment: CENTER-RIGHT;");
```

```
// Add columns to the table
tblBank.setPrefWidth(400);
tblBank.setPrefHeight(700);
tblBank.getColumns().add(tcolName);
tblBank.getColumns().add(tcolId);
tblBank.getColumns().add(tcolBalance);

// Add to the root
root.getChildren().add(tblBank);
```

- Read the data into the underlying data model ArrayList (data)
- Connect the columns of the Table to the attributes of the Customer class
    - A CellValueFactory is a class that knows how to get the value of a cell from the underlying data model class (Customer)
    - The parameter to the constructor is a string which is the name of the attribute to display

```
readData();          // Read the data (Vector of Vectors)
tcolName.setCellValueFactory(
   new PropertyValueFactory<Customer, String>("name"));
tcolId.setCellValueFactory(
   new PropertyValueFactory<Customer, Integer>("id"));
tcolBalance.setCellValueFactory(
   new PropertyValueFactory<Customer, Double>("balance"));
```

- We want the balance to be displayed with 2 places to the right of the decimal, so that 125.3 is displayed as 125.30. To do this, we need a CellFactory (different from a CellValueFactory) for the Balance column
- Override the updateItem method in this Factory to display the balance as desired

```
tcolBalance.setCellFactory(tc -> new TableCell<Customer, Double>() {
   @Override
   protected void updateItem(Double balance, boolean empty) {
      super.updateItem(balance, empty);
      if (empty) {
         setText(null);
      } else {
         setText(String.format("%10.2f", balance));
      }
   }
});
```

- Finally, set the ArrayList (data) as the source for items in the Table

```
tblBank.setItems(data);
```

- **Details about reading the data**
  - Write the readData() method. It should:
    - Open the file as a DataInputStream
    - For each customer, read in the Name (String), id number (int), and account balance (double)
    - Store the id number in an Integer and the balance in a Double. This makes each of these an object and not a primitive type
    - Create a Customer object using the Customer constructor
    - Add the Customer to the ArrayList data
    - On EOF, simply return

Get this program to run and display the data.

Check this out.
Click on the Name label at the top of the label column. What happens? Click it again. What happens now? Try with the other columns. Describe what happened as answer #1 in Lab08Part2.txt.

Does this actually change the underlying ArrayList? How can you check? Explain as answer #2 in Lab08Part2.txt.

Submit your *.java file(s) and the Lab08Part2.txt file to the Lab08 Assignment folder when your program for Part 2 is working correctly.