THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2212b - Winter 2014
Introduction to Software Engineering

Lab 8 - Spreadsheets with JTable

Part I

Introduction to the `JTable` Control

Estimated Time: 60-90 minutes

1 Introduction

The `JTable` control is a standard part of the Swing toolkit, found in the `javax.swing` package. It is a versatile control, allowing tabular data to be displayed and manipulated in a variety of ways, but can be somewhat confusing when one is using it for the first time. This lab will show you the basics of `JTable`: how to display and edit tabular data in a Swing application. Additionally, some more advanced techniques will be covered, such as creating a custom `TableModel`, making cells read-only, and rendering cells using a custom `TableCellRenderer`.

2 Getting Started

1. Change to your individual Git repository, create a `lab8` directory, and change to it:

```
$ cd ~/courses/cs2212/labs
$ mkdir lab8
$ cd lab8
```

2. Create a file `pom.xml` with the following elements:

   - `groupId`: `ca.uwo.csd.cs2212.USERNAME`
   - `artifactId`: `USERNAME-lab8`
   - `version`: `1.0-SNAPSHOT`

   Refer to lab 2 for the full details of creating a `pom.xml` (remember that you will also need a `modelVersion` tag). Of course, `USERNAME` should be replaced with your UWO username in **lower case**.

3. Edit the `pom.xml` file so that the JAR file that it produces is executable (refer to lab 2). Set the main class to `ca.uwo.csd.cs2212.USERNAME.App`.

4. Add the following to your `pom.xml` (inside the `project` tag):

```
<properties>
  <maven.compiler.source>7</maven.compiler.source>
  <maven.compiler.target>7</maven.compiler.target>
</properties>
```

   This ensures that we're using JDK 7 for compilation, since there is some code in this lab that uses Java 7 features. Note that you will most likely need to ensure that your `JAVA_HOME` environment variable is set to the home of a JDK 7 installation on your system (i.e. to the directory that contains the `bin` directory, which, in turn, contains `javac`). Otherwise, Maven will just use the default JDK on your system, which may or may not be JDK 7.

5. Create the directory structure to host our project:

```
$ mkdir -p src/{main,test}/{java,resources}
```

6. Create the directory structure to house your Java package:

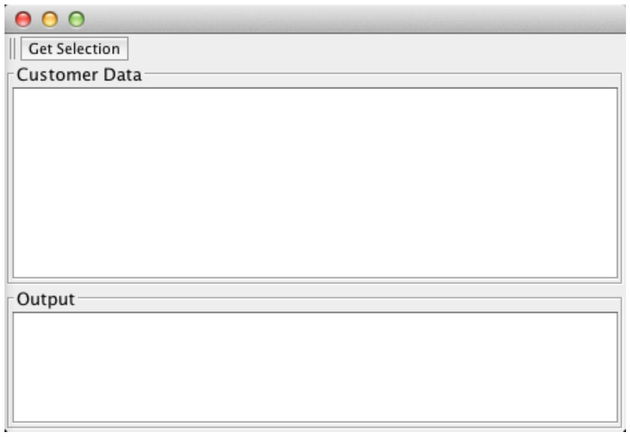
   ```
   $ mkdir -p src/main/java/ca/uwo/csd/cs2212/USERNAME
   ```

   Again, `USERNAME` should be your UWO username in **lowercase**.

7. Download `App.java` and `MainWindow.java` from the following Gist and put them in the proper directory: https://gist.github.com/jsuwo/9129747.

   - Customize the package statements in each file.

8. Package your code as a JAR and run it. You should see the following window appear.



We have a `JTable`, but it's not of much use without data. Let's fix that. We'll assume that we're building a simple application for a company to allow us to view customer information.

3 Adding Some Data to Our `JTable`

We will see a more advanced way of adding data to a table later. For now, we'll use a very simple method to get us started. The `JTable` makes use of the concept of a *table model*. To display data in a `JTable`, we create a table model and add the data to the model. We then associate the table model with the table. This way, the storage of the data is decoupled from its presentation.

In this section, we'll see the use of a `DefaultTableModel`, which simply takes a multi-dimensional array containing the data to be displayed in the table. Later, we'll see how we can create a custom table model.

1. Open the `MainWindow.java` class in your editor.

2. Add a `private`, `void` method `initTable` to the class and call this method from the constructor. We will put all of our table initialization code in this method.

3. Add a `private`, `void` method `initModel` to the class and call this method from `initTable`. Add the following code:

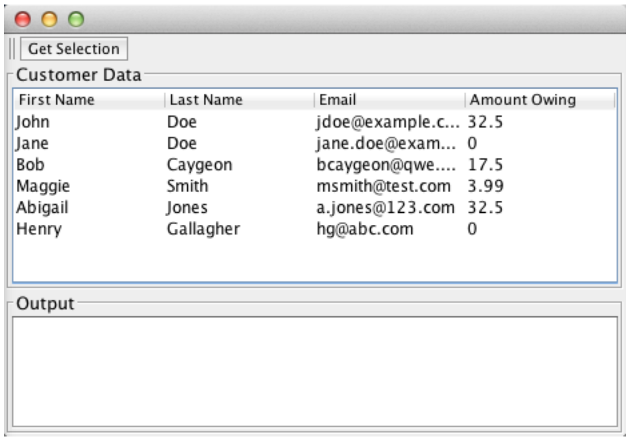
   ```
   tblCustomers.setModel(new DefaultTableModel(
     new Object [][] {
       {"John", "Doe", "jdoe@example.com", 32.50},
       {"Jane", "Doe", "jane.doe@example.net", 0}
     },
     new String [] {
       "First Name", "Last Name", "Email", "Amount Owing"
       }
   ));
   ```

Notice that the first argument to the `DefaultTableModel` constructor is a multi-dimensional array containing the data to be displayed in the table. The second argument is an array of column names.

4. Add at least 3-4 more customers to the model. Be sure to have some customers with positive balances, and some with zero balances.

5. Package your code and run it.

   - Observe that the data is displayed in the table, and the column headers have been set.
   - Notice that we get some functionality right out of the box:
     - Try double-clicking a cell.
     - Try selecting one or more rows.
     - Try rearranging the order of the columns by dragging their column headers.
     - Try resizing the columns.



First Name	Last Name	Email	Amount Owing
John	Doe	jdoe@example.c...	32.5
Jane	Doe	jane.doe@exam...	0
Bob	Caygeon	bcaygeon@qwe....	17.5
Maggie	Smith	msmith@test.com	3.99
Abigail	Jones	a.jones@123.com	32.5
Henry	Gallagher	hg@abc.com	0

So far so good. Still, the email addresses are cut off, and we might like to enable sorting. Let's see if we can address both issues.

4 Column Widths / Reordering / Sorting

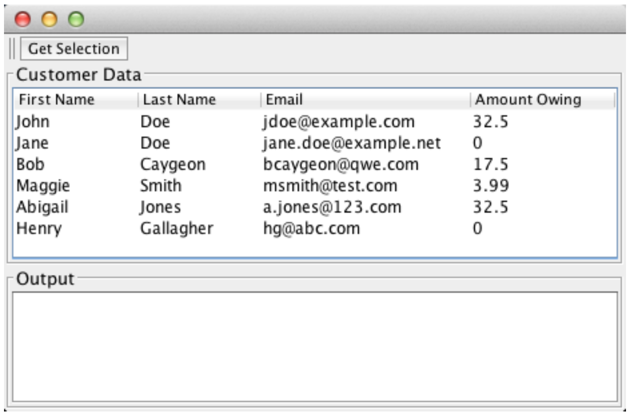
1. Add a `private`, `void` method `setColumnWidths` that is called from `initTable`:

   ```
   tblCustomers.getColumnModel().getColumn(0).setPreferredWidth(35);
   ```

   We first obtain the *column model* from the table, retrieve the first column (column `0`), and then set its *preferred* width to 35.

2. Set the remaining column widths to `35`, `100`, and `50`, respectively.

3. Package and run your code. You should now see the full text of the email addresses.



First Name	Last Name	Email	Amount Owing
John	Doe	jdoe@example.com	32.5
Jane	Doe	jane.doe@example.net	0
Bob	Caygeon	bcaygeon@qwe.com	17.5
Maggie	Smith	msmith@test.com	3.99
Abigail	Jones	a.jones@123.com	32.5
Henry	Gallagher	hg@abc.com	0

4. Try resizing the window. Notice that the relative widths of the columns are preserved.

Next, we might like to allow our users to sort the rows in the table by clicking a column header. While there are more advanced and customizable ways to do this, we can enable basic sorting functionality with a single call.

5. Add the following call to `initTable`:

```
tblCustomers.setAutoCreateRowSorter(true);
```

6. Package and run your code. You should be able to click the column headers to sort the rows of the table.

5 Retrieving Selected Rows

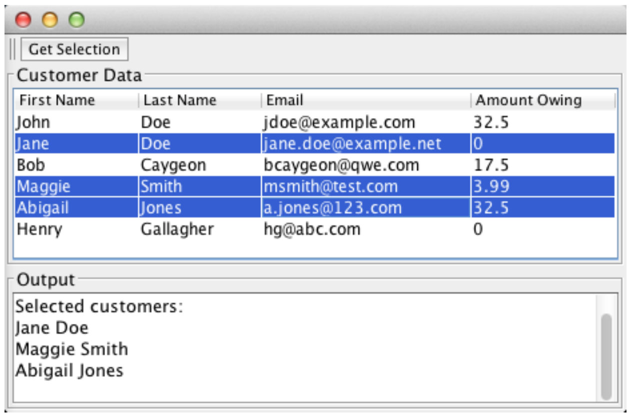
We often want the user to be able to select various rows and then perform some action upon them. In this section, we'll see a simple way of getting the selected rows of the table.

1. Add the following to the `initTable` method:

```
1  this.btnGetSelection.addActionListener(new ActionListener() {
2
3    @Override
4    public void actionPerformed(ActionEvent e) {
5      StringBuilder sb = new StringBuilder();
6      int[] selectedRows = tblCustomers.getSelectedRows();
7
8      sb.append("Selected customers:\n");
9
10     for (int row : selectedRows) {
11       String firstName = tblCustomers.getModel().getValueAt(row, 0).toString();
12       String lastName = tblCustomers.getModel().getValueAt(row, 1).toString();
13       sb.append(firstName).append(" ").append(lastName).append("\n");
14     }
15
16     txtOutput.setText(txtOutput.getText() + "\n" + sb.toString());
17   }
18 });
```

- On line 6, we obtain the indices of all selected rows by calling the `getSelectedRows` method on the `JTable`.
- In lines 10–13, we iterate over each of the selected rows. For each such row, we obtain the row's data by first calling `getModel` and then calling `getValueAt`, passing the row and column index.
- Finally, on line 16, we add the data we retrieved to the output pane at the bottom of the window.
- As an aside, if you are unfamiliar with the syntax used in the `for` loop on line 10, you should Google *Java foreach loop*. It is a very convenient way of iterating over a collection – much more so than the standard `for` loop.

2. Package and run your code. Try selecting a customer (and multiple customers) and clicking the **Get Selection** button in the toolbar.



First Name	Last Name	Email	Amount Owing
John	Doe	jdoe@example.com	32.5
Jane	Doe	jane.doe@example.net	0
Bob	Caygeon	bcaygeon@qwe.com	17.5
Maggie	Smith	msmith@test.com	3.99
Abigail	Jones	a.jones@123.com	32.5
Henry	Gallagher	hg@abc.com	0

```
Selected customers:
Jane Doe
Maggie Smith
Abigail Jones
```

6 Creating a Custom Table Model

The table model that we have used so far works for displaying simple data, but we need to consider how we are going to store that data when changes are made to it in the table. Suppose we had a `Customer` class to store our customers' data. In order to display our `Customer` objects in the table, we would have to retrieve the data from them by calling their accessor methods (e.g. `getFirstName`, `getLastName`), put all of these values in a multi-dimensional array as we did earlier, and initialize a `DefaultTableModel`. Then, we would need to keep track of changes to the data in the table, and then update our `Customer` objects accordingly. This seems cumbersome.

Ideally, we'd like to have our table model directly linked to our `Customer` objects so that they can be updated as soon as the data in the table is changed. In this section, we'll see one way of doing this.

1. Download the `Customer.java` and `CustomerTableModel.java` classes from the following Gist and put them in the appropriate directory: https://gist.github.com/jsuwo/9129747.
   - Customize the package statements in each file.

The `Customer` class is straightforward, so let's focus on the `CustomerTableModel` class.

```
 7 public class CustomerTableModel extends AbstractTableModel {
 8
 9     private final static int COLUMN_COUNT = 4;
10     private final static int IDX_FIRST_NAME = 0;
11     private final static int IDX_LAST_NAME = 1;
12     private final static int IDX_EMAIL = 2;
13     private final static int IDX_BALANCE = 3;
14
15     private final List<Customer> customers;
16
17     public CustomerTableModel() {
18         customers = new ArrayList<>();
19     }
```

On line 7, we first declare a class extending from `AbstractTableModel`. This class provides some basic functionality for managing a table model, such as the ability to add and notify *listeners* when the data in the model changes.

Since we want to work with `Customer` objects directly in the table model, we declare a `List` to store them on line 15. We could, of course, use any data structure we liked, but a `List` will do just fine here.

Finally, in line 18, we instantiate the list. Notice that we're using the *diamond operator*, which is new to Java 7. This operator reduces the amount of typing we have to do when working with generics.

```
21 public List<Customer> getCustomers() {
22   return customers;
23 }
24
25 @Override
26 public int getRowCount() {
27   return customers.size();
28 }
29
30 @Override
31 public int getColumnCount() {
32   return COLUMN_COUNT;
33 }
```

Next, we declare a `getCustomers` method in line 21. This will allow us to easily retrieve customers later, such as when we need to retrieve all customers that were selected in the table.

The `getRowCount` and `getColumnCount` methods provide information to the table on the number of rows and columns to display, respectively.

```
35 @Override
36 public Class<?> getColumnClass(int columnIndex) {
37   return (columnIndex == IDX_BALANCE ? Double.class : String.class);
38 }
```

The `getColumnClass` method declares the type of data stored in each column. We'll see why this is useful later when we cover cell rendering. For now, just observe that we'll store data of type `Double` in the *Balance Owing* column, and data of type `String` in the rest of the columns.

```
40 @Override
41 public String getColumnName(int columnIndex) {
42   switch (columnIndex) {
43     case IDX_FIRST_NAME:
44       return "First Name";
45     case IDX_LAST_NAME:
46       return "Last Name";
47     case IDX_EMAIL:
48       return "Email";
49     case IDX_BALANCE:
50       return "Balance Owing";
51     default:
52       return null;
53     }
54 }
```

We then declare a `getColumnName` method in lines 41–54, which returns the column names that will be used in the column headers of the table.

Next, we need to provide the implementation for the `getValueAt` method, which will be called for each *(row, column)* pair in the table to retrieve the value in each table cell. The implementation has been started for you, providing some validation in case an invalid `rowIndex` is specified.

```
56 @Override
57 public Object getValueAt(int rowIndex, int columnIndex) {
58   if ((rowIndex < 0) || (rowIndex >= customers.size()))
59     return null;
60 
61     // add implementation here
62 }
```

2. Add the following logic to the `getValueAt` method:

    - Retrieve the appropriate `Customer` object from the `customers` list (use the `rowIndex`)
    - Use a `switch` statement as in the `getColumnName` method to return the appropriate data from the `Customer` object depending on the specified `columnIndex`.
        - If an invalid `columnIndex` is specified (e.g. the `default` condition), return `null`.

Now that we have `getValueAt` implemented, we need to implement `setValueAt`, which will be called when the data in a table cell is edited. This method will need to store the changed data in the appropriate `Customer` object. Once again, the implementation has been started for you:

```
64 @Override
65 public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
66   if ((rowIndex < 0) || (rowIndex >= customers.size()) || columnIndex != IDX_BALANCE)
67     return;
68 
69     // add implementation here
70     }
```

Let's assume that we're developing this application for the billing department of the company, and we don't want to be able to edit a customer's personal information – we only want to be able to edit a customer's balance.

3. Add the following logic to the `setValueAt` method:

    - Retrieve the appropriate `Customer` object from the `customers` list (use the `rowIndex`)
    - If the `columnIndex` specified represents the *Balance Owing* column:
        - Update the customer's balance (you'll need to cast `aValue` appropriately)
        - Call `fireTableCellUpdated(rowIndex, columnIndex);`

The `fireTableCellUpdated` method is provided by the `AbstractTableModel` superclass, and takes care of notifying external components of the change, if they have subscribed to `tableChanged` events, which we'll discuss later.

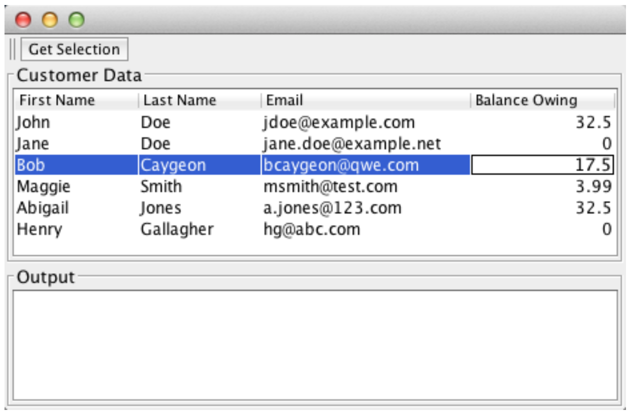
Lastly for this class, we need to tell the `JTable` which columns are editable and which are not. The `isCellEditable` method is called with a `rowIndex` and `columnIndex`, and must return a Boolean value indicating whether or not the cell can be edited.

```
@Override
public boolean isCellEditable(int rowIndex, int columnIndex) {
  // implement me
}
```

4. Implement the `isCellEditable` method so that it returns `true` if the `columnIndex` passed to the method is the *Balance Owing* column, and `false` otherwise.

Now that we've implemented a custom table model, we need to modify our `initModel` method in the `MainWindow` class to use this model instead of a `DefaultTableModel`.

5. Modify the `initModel` method in `MainWindow` as follows:
   - Instantiate a new `CustomerTableModel`
   - Create a number of `Customer` objects (use the same data you used earlier)
   - Add the `Customer` objects to the model (use the `getCustomers` method)
   - Set the `tblCustomers` table to use the model (use the `setModel` method)
6. Package and run your code. Ensure that your `Customer` objects are properly displayed in the table, and that only the *Balance Owing* column can be edited.



First Name	Last Name	Email	Balance Owing
John	Doe	jdoe@example.com	32.5
Jane	Doe	jane.doe@example.net	0
Bob	Caygeon	bcaygeon@qwe.com	17.5
Maggie	Smith	msmith@test.com	3.99
Abigail	Jones	a.jones@123.com	32.5
Henry	Gallagher	hg@abc.com	0

Notice, as well, that the *Balance Owing* column is now right-aligned. This is because we set the class of the column to `Double` in the `getColumnClass` method in `CustomerTableModel`. By default, the `JTable` right-aligns columns of type `Double`.

7 Implementing a Custom Renderer

We've now implemented a custom table model that allows a user to edit only the *Balance Owing* column. Still, the *Balance Owing* column leaves a bit to be desired. First, it displays no currency symbol, which might confuse some users. Additionally, we might also like to highlight which customers have balances owing by display their balances in a different colour. To address these issues, we'll need to implement a custom *renderer* for the column.

1. Download `CurrencyCellRenderer.java` from the following Gist and put it in the proper directory: https://gist.github.com/jsuwo/9129747.
   - Customize the package statement in the file.

The `CurrencyCellRenderer` class extends the `DefaultTableCellRenderer` class, which, in turn, extends the `JLabel` class. `DefaultTableCellRenderer` also implements the `TableCellRenderer` interface. Any class that implements this interface can be used as a renderer for cells in a `JTable`. Since `DefaultTableCellRenderer` extends `JLabel`, it effectively embeds a `JLabel` in any cell it is used to render.

If we wanted to render a cell using a different type of component, we would create a class that extends that component and implements the `TableCellRenderer` interface. For instance, if we wanted to create a renderer to render cells using a `JComboBox`, we might use the following:

```
public class ComboBoxRenderer extends JComboBox implements TableCellRenderer
```

For our purposes here, however, the `DefaultTableCellRenderer` will do just fine as our superclass.

```
11 public class CurrencyCellRenderer extends DefaultTableCellRenderer {
12
13     private final Format formatter;
14
15     public CurrencyCellRenderer() {
16         this.formatter = NumberFormat.getCurrencyInstance();
17
18         // Set horizontal alignment here
19     }
```

The constructor initializes a `Format` instance, which will be used to format a `double` value with a currency symbol.

```
21     @Override
22     public Component getTableCellRendererComponent(JTable table, Object value,
23       boolean isSelected, boolean hasFocus, int row, int column) {
24
25         // Cast the value to a double here.
26         // If the value is > 0, set the foreground color of the cell to red;
27         // otherwise, set it to black.
28
29         // Format the value with a currency symbol
30         String formattedValue = formatter.format(amount);
31
32         // Set the value here
33
34         return this;
35     }
36 }
```

Next, we have the `getTableCellRendererComponent` method. This method is called for every cell in the table to which the renderer has been applied. It is passed information such as the value of the cell (passed as an `Object`), whether or not the cell is selected and/or focused, and the row and column indices of the cell.

This method returns a `Component`, which is an ancestor of all the Swing controls. Hence, we can return any control we like, and this control will be displayed in the table cell being rendered. Since `JLabel` is an ancestor of our class, we can simply return `this`, and the cell will be rendered as a `JLabel`.

2. Replace line 25 with an expression that casts the value passed to the method as a double and stores it in an `amount` variable.
3. Replace lines 26 and 27 with code to set the foreground colour of the label based on the amount in the table cell:
   - If the amount is positive (greater than zero), set the colour to red.
   - Otherwise, set the amount to black.
   - Look up the `setForeground` method, which is declared in the `JComponent` class (an ancestor of our class).
   - Look up the `java.awt.Color` class.
4. Replace line 32 with a call to the superclass' `setValue` method. Set its value to `formattedValue`, which will ensure that the value displayed in the table cell starts with a currency symbol.

We've now implemented our renderer, but still haven't applied it to any cells in our table.

5. Edit `MainWindow.java` and add the following call to the `initTable` method:

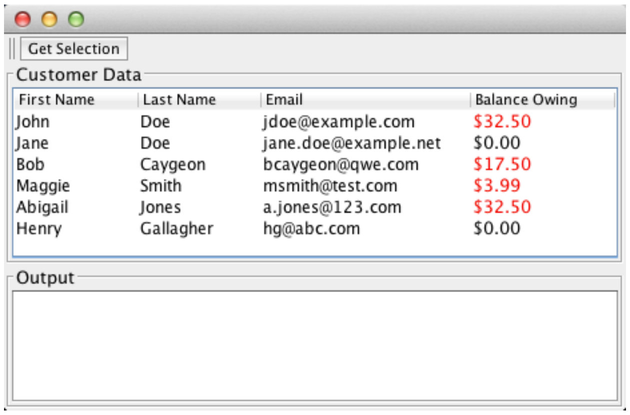
```
tblCustomers.setDefaultRenderer(Double.class, new CurrencyCellRenderer());
```

This indicates to the `JTable` that we wish to use the `CurrencyCellRenderer` for any columns whose data is of type `Double`. Recall the `getColumnClass` method from the `CustomerTableModel` class:

```
@Override
public Class<?> getColumnClass(int columnIndex) {
  return (columnIndex == IDX_BALANCE ? Double.class : String.class);
}
```

We specified the class of the *Balance Owing* column to be `Double`. Hence, by setting the default renderer for all columns of type `Double` to be `CurrencyCellRenderer`, we ensure it will be used for all cells in the *Balance Owing* column.

6. Package and run your code. Verify that the *Balance Owing* column displays red text for positive balances and black text for zero balances.



First Name	Last Name	Email	Balance Owing
John	Doe	jdoe@example.com	$32.50
Jane	Doe	jane.doe@example.net	$0.00
Bob	Caygeon	bcaygeon@qwe.com	$17.50
Maggie	Smith	msmith@test.com	$3.99
Abigail	Jones	a.jones@123.com	$32.50
Henry	Gallagher	hg@abc.com	$0.00

Notice, however, that the text in the column is once again left-aligned. This is because, previously, we were using the default cell renderer for the column, which right-aligns values of type `Double`. Now that we're using a custom renderer, we have to handle the alignment manually.

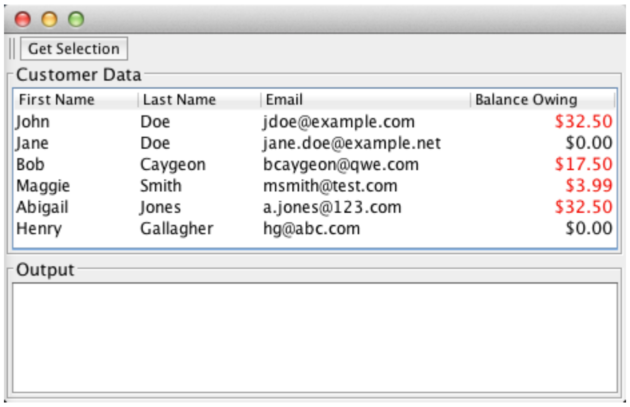
Since `JLabel` is one of the ancestors of our class, we can call the `setHorizontalAlignment` method to right-align the value in the label.

7. Look up the `setHorizontalAlignment` method in the `JLabel` class and figure out how to right-align the text in a `JLabel`.

8. Add the necessary line to right-align the text of the label in the `CurrencyCellRenderer` constructor.

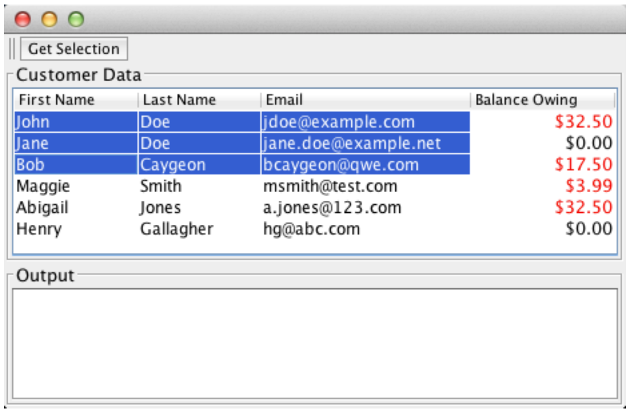
   - Recall that `JLabel` is an ancestor of our class, so we have its methods available to us.

9. Package and run your code. Verify that the text in the *Balance Owing* column is now right-aligned.



First Name	Last Name	Email	Balance Owing
John	Doe	jdoe@example.com	$32.50
Jane	Doe	jane.doe@example.net	$0.00
Bob	Caygeon	bcaygeon@qwe.com	$17.50
Maggie	Smith	msmith@test.com	$3.99
Abigail	Jones	a.jones@123.com	$32.50
Henry	Gallagher	hg@abc.com	$0.00

We still have one more problem to address. If you try selecting a row in the table, the *Balance Owing* column does not appear to be selected. This is because we have to specify the background colour of the cell in `CurrencyCellRenderer` when the cell is selected.



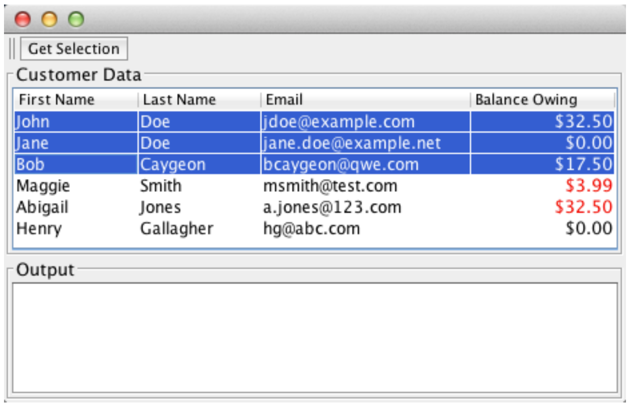
First Name	Last Name	Email	Balance Owing
John	Doe	jdoe@example.com	$32.50
Jane	Doe	jane.doe@example.net	$0.00
Bob	Caygeon	bcaygeon@qwe.com	$17.50
Maggie	Smith	msmith@test.com	$3.99
Abigail	Jones	a.jones@123.com	$32.50
Henry	Gallagher	hg@abc.com	$0.00

10. Add the following code to the `getTableCellRendererComponent` method in `CurrencyCellRenderer` **AFTER** the existing code that sets the foreground colour of the text:

```
if (isSelected) {
  setForeground((Color)UIManager.get("Table.selectionForeground"));
  setBackground((Color)UIManager.get("Table.selectionBackground"));
}
else {
  setBackground((Color)UIManager.get("Table.background"));
}
```

    Note that you will need to import the `javax.swing.UIManager` class. This code checks if the cell is currently selected. If so, it sets the foreground and background colours of the cell according to the default colours for a selected table cell. Otherwise, it sets the background of the cell to the default colour for an unselected table cell (you've already set the foreground colour for the cell above).

11. Package and run your code. Ensure that the colours now look correct in the *Balance Owing* column when you select a row.



First Name	Last Name	Email	Balance Owing
John	Doe	jdoe@example.com	$32.50
Jane	Doe	jane.doe@example.net	$0.00
Bob	Caygeon	bcaygeon@qwe.com	$17.50
Maggie	Smith	msmith@test.com	$3.99
Abigail	Jones	a.jones@123.com	$32.50
Henry	Gallagher	hg@abc.com	$0.00

8 Finishing Touches

In this section, we'll spend just a moment putting a few finishing touches on the table.

First, we are currently using *row selection mode*: when we select any cell in the table, the entire row in which the cell is located will be selected. If we're developing a spreadsheet-like application, we might want to switch to *cell selection mode*.

Next, power users often prefer to use the keyboard for reasons of speed and efficiency. By default, when a cell is selected in the table and the user presses Enter, the selection moves downward. Instead, we might like to start editing the currently selected cell.

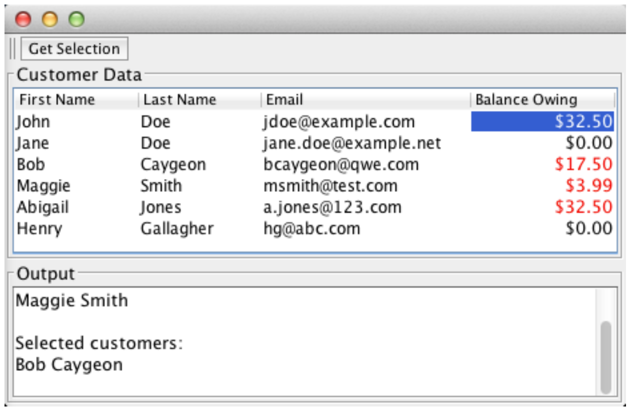
We'll see how to address these issues in this section.

1. Edit the `initTable` method in `MainWindow` and add the following line:

```
tblCustomers.setCellSelectionEnabled(true);
```

   This switches the table to cell selection mode, so that individual cells can be selected, rather than entire rows.

2. Package and run your code. Ensure that you can select individual cells.



First Name	Last Name	Email	Balance Owing
John	Doe	jdoe@example.com	$32.50
Jane	Doe	jane.doe@example.net	$0.00
Bob	Caygeon	bcaygeon@qwe.com	$17.50
Maggie	Smith	msmith@test.com	$3.99
Abigail	Jones	a.jones@123.com	$32.50
Henry	Gallagher	hg@abc.com	$0.00

```
Maggie Smith

Selected customers:
Bob Caygeon
```

3. Try selecting a cell and pressing Enter. Notice that the selection moves to the next cell below.

4. Add the following line to the `initTable` method:

```
tblCustomers.getInputMap().put(KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0), "startEditing");
```

   Here, we're changing the *input map* of the table so that when the user presses Enter, we start editing the currently selected cell.

5. Package and run your code. Try selecting a cell in the *Balance Owing* column and pressing Enter. The cell should enter "edit mode".

9 JTable Resources

There are plenty of things that one can do with a `JTable` that we were not able to cover in this lab. For more information on this control, see http://docs.oracle.com/javase/tutorial/uiswing/components/table.html.

10 Submitting Your Lab

- Take a screenshot of your running application.
- Name the screenshot `lab8.png` and place it in your `lab8` directory.
  - Note: it **must** be named `lab8.png` – not `lab8.PNG`, `lab8.png.png`, etc.
- Commit your code and push to GitHub.
- To submit your lab, create the tag `lab8` and push it to GitHub. For a reminder on this process, see Lab 1.

Note that, for this lab, due to the difficulty in automatically testing GUI programs (it is possible, but more difficult), the automarker will not actually be running any tests on your lab. Instead, it will ensure your code compiles, and will then automatically mark your lab as complete. We will then later test your lab and will notify you if there are any problems with your lab that might require resubmission.