7 JavaFX Tables and Client GUI

ORACLE

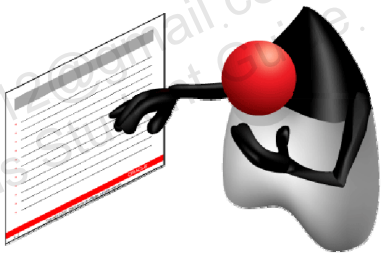
Objectives

After completing this lesson, you should be able to:

- Create a table and custom table cell
- Apply CSS to a table
- Recognize JavaFX development practices
- Describe the BrokerTool application interface
- Identify the JavaFX components and charts to use in the BrokerTool interface

Topics

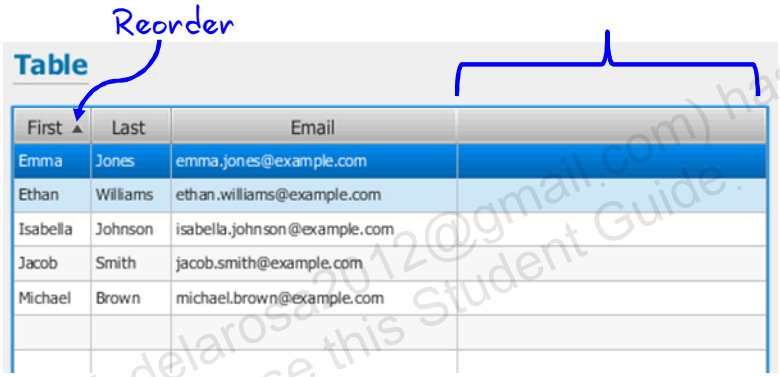
- Develop a table and custom table cell
- Add CSS to a table
- Describe the BrokerTool application interface



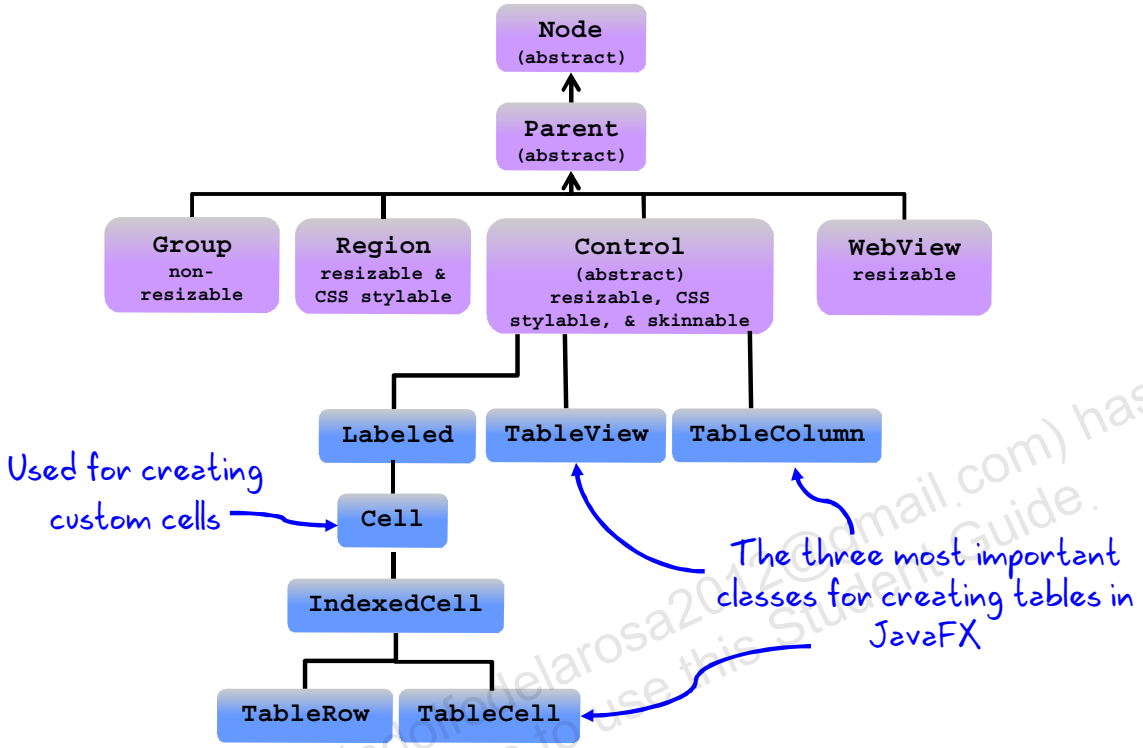
Tables in JavaFX

Tables in JavaFX are very powerful and support the following:

- Column reordering by user
- Multiple column sorting
- Width resizing
- Cell factories for customizing cell content



TableView is a Node



Several classes in the JavaFX SDK API are designed to represent data in a tabular form. The most important classes for creating tables in JavaFX applications are `TableView`, `TableColumn`, and `TableCell`. You can populate a table by implementing the data model and by applying a cell factory. The table classes provide built-in capabilities to sort data in columns and to resize columns when necessary. The `TableView` control has a number of features, including:

- `TableColumn` API:
  - Support for cell factories to easily customize cell contents in both rendering and editing states
  - Specification of `minWidth/ prefWidth/maxWidth` and also fixed-width columns
  - Width resizing by the user at runtime
  - Column reordering by the user at runtime
  - Built-in support for column nesting
- Different resizing policies to determine what happens when the user resizes columns
- Support for multiple column sorting by clicking the column header (press the Shift key while clicking a header to sort by multiple columns)

Creating a Table

`TableView` and `TableColumn` are the minimum classes required to create a table.

- `TableView<S> S`: The type of the objects contained in the `TableView` items list
- `TableColumn<S,T>`
  - `S`: The type of the `TableView` generic type (that is, `S` == `TableView<S>`)
  - `T`: The type of content in all cells in this `TableColumn`

The next slide shows the code example.

A `TableView` is made up of a number of `TableColumn` instances. Each `TableColumn` in a table is responsible for displaying (and editing) the contents of that column. In addition to being responsible for displaying and editing data for a single column, a `TableColumn` also contains the necessary properties to:

- Be resized (using `minWidth/prefWidth/maxWidth` and fixed-width properties)
- Have its `visibility` toggled
- Display `header text`
- Display any `nested columns` it may contain
- Have a `context menu` when the user right-clicks the column header area
- Have the contents of the table be sorted (using `comparator`, `sortable`, and `sortType`)

When you create a `TableColumn` instance, perhaps the two most important properties to set are the column `text` (what to show in the column header area) and the column `cell value factory` (which is used to populate individual cells in the column).

The entire code sample is displayed in the following slide.

Creating a Table: Code Example

```
public class Person {
 private StringProperty firstName;
 public void setFirstName(String value) { firstNameProperty().set(value); }
 public String getFirstName() { return firstNameProperty().get(); }
 public StringProperty firstNameProperty() {
   if (firstName == null) firstName = new SimpleStringProperty(this, "firstName");
   return firstName;
 }
 private StringProperty lastName;
 public void setLastName(String value) { lastNameProperty().set(value); }
 public String getLastName() { return lastNameProperty().get(); }
 public StringProperty lastNameProperty() {
    if (lastName == null) lastName = new SimpleStringProperty(this, "lastName");
    return lastName;
 }
}
...
TableView<Person> table = new TableView<Person>();
TableColumn<Person,String> firstNameCol
   = new TableColumn<Person,String>("First Name");
…
table.getColumns().setAll(firstNameCol, lastNameCol);
```

TableCell<S,T>

- Represents a single row/column intersection in a `TableView`
- Contains the following properties:
  - `tableColumn:` The `TableColumn` instance that backs this `TableCell`
  - `tableView:` The `TableView` associated with this `TableCell`
  - `tableRow:` The `TableRow` in which this `TableCell` is currently placed

To represent this intersection, a TableCell contains an index property, as well as a tableColumn property. In addition, a TableCell instance knows what TableRow it exists in.

Cell<T>

- Is used for an individual cell in a `TableView`
- Every cell is associated with a single data item represented by the `item` property.
- A cell is responsible for rendering any item that resides within it, which is usually `text`.
- A cell is a control and is essentially a "model" (in MVC terms).
- Enables customization by using a cell factory

The `Cell` API is used for virtualized controls such as `ListView`, `TreeView`, and `TableView`. A `Cell` is a `Labeled` Control, and is used to render a single “row” inside a `ListView`, `TreeView` or `TableView`. Cells are also used for each individual “cell” inside a `TableView` (that is, each row/column intersection).

Note: For details, see the JavaDoc for each individual control.

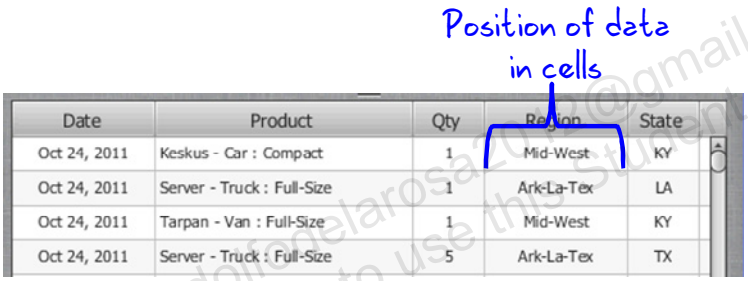
Every `Cell` is associated with a single data item (represented by the item property). The `Cell` is responsible for rendering that item and, where appropriate, for editing the item. An item within a `Cell` can be represented by text or some other control such as a `CheckBox`, `ChoiceBox`, or any other `Node` such as an `HBox`, `GridPane`, or even a `Rectangle`.

Because `TreeView`, `ListView`, `TableView`, and other such controls can potentially be used for displaying extremely large amounts of data, it is not practical to create an actual `Cell` for every item in the control. You can represent extremely large data sets by using very few Cells. Each `Cell` is “recycled,” or reused, which makes this control virtualized.

Because `Cell` is a `Control`, it is essentially a “model.” Its skin is responsible for defining the look and layout, while the Behavior is responsible for handling all input events and using that information to modify the control state. Also, the `Cell` is styled from CSS just like any other control. However, it is not necessary to implement a skin for most uses of a `Cell`.

Cell Factory

- To specialize the `Cell` used for the `TableView`, you must provide an implementation of the `cellFactory` callback function defined on the `TableView`.
- The cell factory is called by the platform whenever it determines that a new cell needs to be created.
- The implementation of the cell factory is responsible for creating a `Cell` instance and also configuring that `Cell` so that it reacts to changes in its state.



It is the responsibility of the various virtualized containers' skins to render the default representation of the `Cell` item. For example, the `ListView` by default converts the item to a String and calls `Labeled.setText(java.lang.String)` with this value. If you want to specialize the `Cell` used for the `ListView` (for example), then you must provide an implementation of the `cellFactory` callback function defined on the `ListView`. A similar API exists on most controls that use Cells (for example, `TreeView`, `TableView`, `TableColumn`, and `ListView`).

The cell factory is called by the platform whenever it determines that a new cell needs to be created. For example, perhaps your ListView has 10 million items. Creating all 10 million cells would be prohibitively expensive. So instead, the `ListView` skin implementation might only create just enough cells to fit the visual space. If the `ListView` is resized to be larger, the system will determine that it needs to create some additional cells. In this case, it will call the `cellFactory` callback function (if one is provided) to create the `Cell` implementation that should be used. If no cell factory is provided, the built-in default implementation will be used.

The implementation of the cell factory is then responsible not just for creating a `Cell` instance, but also for configuring that `Cell` so that it reacts to changes in its state. In the example in the slide, the cell data is positioned vertically and horizontally within the cell using the `textAlignmentProperty` inherited from the `Labeled` class.

Custom Cell Factory: FXML Example

```
<TableView fx:id="salesTable" >
    <columns>
        <TableColumn text="Date" prefWidth="100">
           <cellValueFactory><PropertyValueFactory
    property="date" /></cellValueFactory>
           <cellFactory>
               <FormattedTableCellFactory alignment="center">
                     <format><DateFormat
    fx:factory="getDateInstance"/></format>
               </FormattedTableCellFactory>
           </cellFactory>
        </TableColumn>
        <TableColumn text="Product" prefWidth="180">
           <cellValueFactory><PropertyValueFactory
    property="productId" /></cellValueFactory>
           <cellFactory><FormattedTableCellFactory
    alignment="left"/></cellFactory>
        </TableColumn>
```


This example represents cellFactory alignment for the columns.

Custom Cell: Example

```
public class FormattedTableCellFactory<S,T> implements
   Callback<TableColumn<S,T>, TableCell<S,T>> {
   private TextAlignment alignment;
   private Format format;

   public TextAlignment getAlignment() {
       return alignment;
   }

   public void setAlignment(TextAlignment alignment) {
       this.alignment = alignment;
   }

   public Format getFormat() {
       return format;
   }

   public void setFormat(Format format) {
       this.format = format;
   }
```


This example represents an implementation of the Callback class and creates instances of the TextAlignment and Format classes using these parameters:

- S: The type of the TableView generic type (that is, S == TableView<S>)
- T: The type of the content in all cells in this TableColumn

Table Data Model

`ObservableList` is the underlying data model for the `TableView`.

```
ObservableList<Person> teamMembers = getTeamMembers();
 table.setItems(teamMembers);

 TableColumn<Person,String> firstNameCol =
   new TableColumn<Person,String>("First Name");
   firstNameCol.setCellValueFactory
   (new PropertyValueFactory("firstName"));
 TableColumn<Person,String> lastNameCol =
   new TableColumn<Person,String>("Last Name");
   lastNameCol.setCellValueFactory
   (new PropertyValueFactory("lastName"));

 table.getColumns().setAll(firstNameCol, lastNameCol);
```

ORACLE

The `TableView` instance is defined as follows:

```
TableView<Person> table = new TableView<Person>();
```

With the basic table defined, look at the data model. This example uses an `ObservableList`. You can set a list directly into the `TableView`:

```
ObservableList<Person> teamMembers = getTeamMembers();
 table.setItems(teamMembers);
```

With the items set, `TableView` automatically updates whenever the `teamMembers` list changes. If the items list is available before the `TableView` is instantiated, it is possible to pass it directly into the constructor.

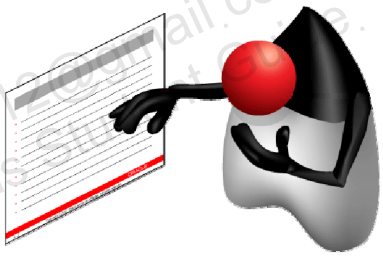
At this point, you have a `TableView` connected to observe the `teamMembers observableList`. The missing ingredient now is the means of splitting out the data contained within the model and representing it in one or more `TableColumn` instances. To create a two-column `TableView` to show the `firstName` and `lastName` properties, you extend the preceding code sample as follows:

```
ObservableList<Person> teamMembers = ...;
 table.setItems(teamMembers);
 TableColumn<Person,String> firstNameCol = new
TableColumn<Person,String>("First Name");
 firstNameCol.setCellValueFactory(new
PropertyValueFactory("firstName"));
 TableColumn<Person,String> lastNameCol = new
TableColumn<Person,String>("Last Name");
 lastNameCol.setCellValueFactory(new
PropertyValueFactory("lastName"));
 table.getColumns().setAll(firstNameCol, lastNameCol);
```

With this code, you have fully defined the minimum properties required to create a `TableView` instance. Running the code results in a `TableView` displayed with two columns: `firstName` and `lastName`. No other properties of the `Person` class are shown because no `TableColumns` are defined.

Topics

- Develop a table and custom table cell
- Add CSS to a table
- Describe the BrokerTool application interface



Benefits of Using CSS with Tables

The benefits of using CSS to style a table (or other control):

- Both time-efficient and memory-efficient for large data sets
- Easy to build and use libraries for custom cells
- Easy to customize cell visuals
- Easy to customize display formatting (for example, 12.34 as $12.34 or 1234%)
- Easy to extend for custom visuals
- Easy to have "panels" of data for the visuals
- Easy to animate the cell size or other properties

CSS and Tables

Use CSS to set cell colors:

- Each cell can be styled directly from CSS. To change the default background of every cell in a TableView to white, you can use the following CSS:

```
.table-cell {
  -fx-padding: 3 3 3 3;
  -fx-background-color: white;
}
```

- To set the color of selected TableView cells to blue, you can add this to your CSS file:

```
.table-cell:selected {
  -fx-background-color: blue;
}
```


Each Cell can be styled directly from CSS.

For `table-cell:selected` to work, you must have `cellSelectionEnabled` set to `true`.

Most Cell implementations extend from `IndexedCell` rather than `Cell`. `IndexedCell` adds two other pseudoclass states: "odd" and "even." With these, you can obtain alternate row striping by using something like the following in your CSS file:

```
.table-cell:odd {
   -fx-background-color: grey;
}
```

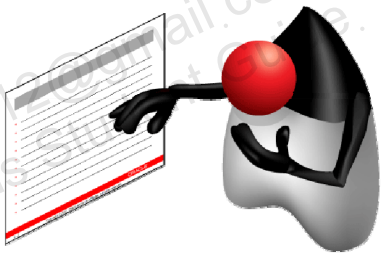
None of these examples requires code changes. Simply update your CSS file to alter the colors. You can also use "hover" and other pseudoclasses in CSS (as with other controls).

One approach to formatting a list of numbers is to use style classes. Suppose you have an `ObservableList` of Numbers to display in a `ListView` and you want to color all of the negative values red and all positive or 0 values black. One way to achieve this is with a custom `cellFactory` that changes the `styleClass` of the `Cell` based on whether the value is negative or positive. This is as simple as adding code to test if the number in the cell is negative, and adding a "negative" `styleClass`. If the number is not negative, the "negative" string should be removed. With this approach, the colors can be defined from CSS, thus enabling simple customization. The CSS file would include something like the following:

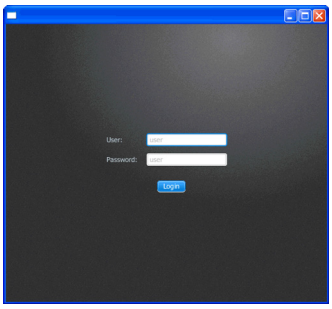
```
.table-cell {
   -fx-text-fill: black;
}
.table-cell .negative {
   -fx-text-fill: red;
}
```

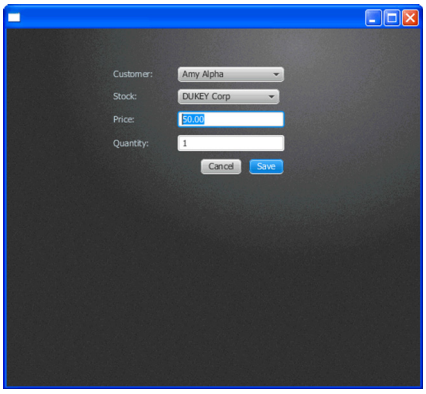
Topics

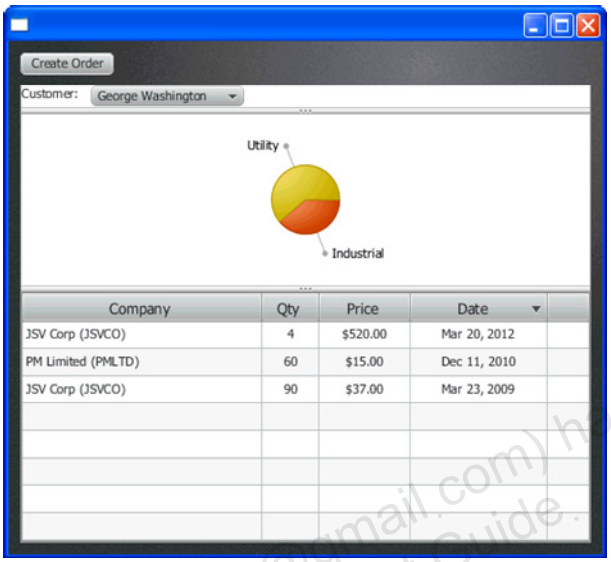
- Develop a table and custom table cell
- Add CSS to a table
- Describe the BrokerTool application interface



BrokerTool Application







- Login window
- Main window
- Order form

JavaFX Development Practices

- Avoid mutable properties that are also updated by the skin.
- Use POJOs as the control's model whenever possible.
- To define style, you should choose CSS rather than explicit API.
- Try to be "deceptively simple" and emphasize content rather than graphics.

Using POJOs as the control's model helps keep the processes separate from the UI.

When you choose CSS rather than explicit API, it is much easier to maintain the CSS styling if you have to make changes later.

Keep the graphics simple and intuitive so the UI is easier for the user to understand.

Application in MVC Terms

- **Model:** JPA
  - Contains the Broker, Customer, Shares, and Stock classes
- **View:** FXML files
- **Controller:** Java files

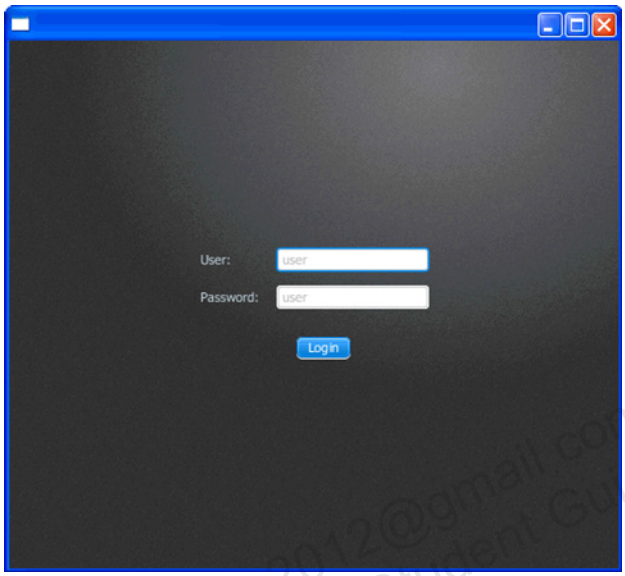
The BrokerTool application follows the Model View Controller design pattern. The model consists of the JPA model, the view is all of the FXML classes, and the controllers are all of the Java classes.

Login Window

View: `login.fxml`

Controls

- Label
- Button
- TextField



The login window is created in `login.fxml`.

- `login.java`
- `AuthenticationException.java`
- `AnimatedPageView.java`

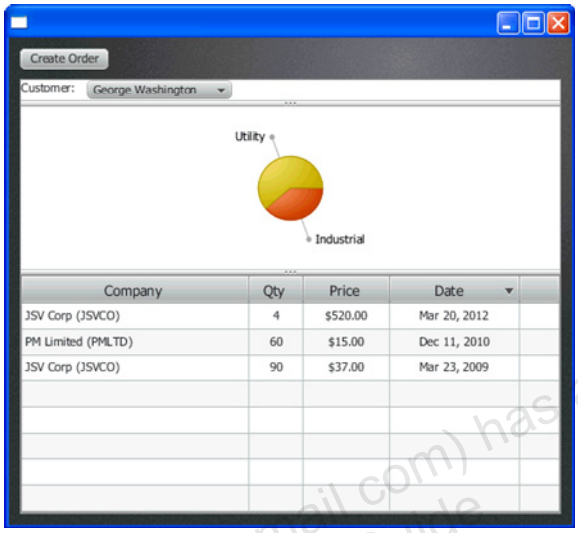
Main Window

View

- BrokerDashboard.fxml
- BrokerToolClient.fxml
- BrokerToolTop.fxml

Controller

- BrokerDashboard.java
- BrokerToolClient.java
- BrokerToolTop.java
- BrokerToolClientApp.java
- …



The BrokerTool application main view is created in a combination of three classes:

- BrokerDashboard.fxml
- BrokerToolClient.fxml
- BrokertToolTop.fxml

The controllers are created in several classes, including:

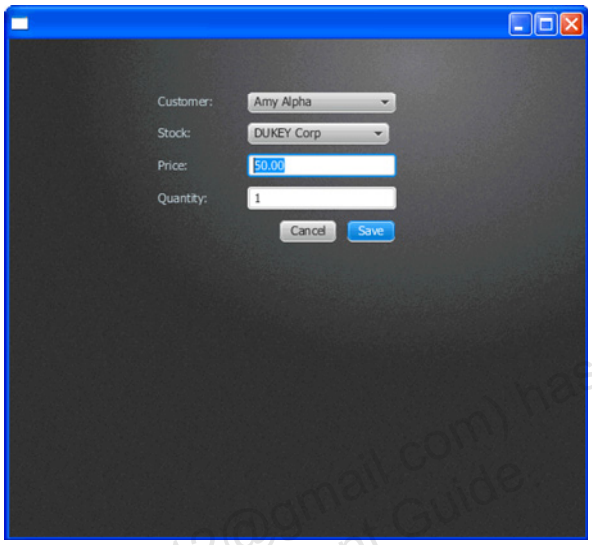
- BrokerDashboard.java
- BrokerToolClient.java
- BrokertToolTop.java
- BrokerToolClientApp.java
- GetSharesTask.java
- GetSharesService.java
- FormattedTableCellFactory.java
- StockTableCellFactory.java
- AnimatedPageView.java

Order Form Window

View: `OrderForm.fxml`

Controls

- Label
- Button
- ChoiceBox
- TextField



The `OrderForm.fxml` class is the view for the Order Form. The controllers for `OrderForm.fxml` include:

- `OrderForm.java`
- `AnimatedPageView.java`
- `BrokerToolClient.java`
- `BrokerToolClientApp.java`
- `GetSharesService.java`
- `GetSharesTask.java`

Quiz

Which are considered the three most important classes for creating tables in JavaFX?

a. `TableView`
b. `TableColumn`
c. `TableCell`
d. `TableRow`
e. `Labeled`

Answer: a, b, c

Summary

In this lesson, you should have learned how to:

- Create a table and custom table cell
- Apply CSS to a table
- Recognize JavaFX development practices
- Describe the BrokerTool application interface
- Identify the JavaFX components and charts to use in the BrokerTool interface

Practice 7: Overview

- 7-1: Creating a Simple Table View
- 7-2: Styling a Smart Table
- 7-3: Creating a Complete BrokerTool Interface