Java SE 7: Develop Rich Client Applications

Activity Guide

D67230GC10
Edition 1.0
May 2012
D77355

ORACLE

Authors

Michael J. Williams, Paromita Dutta, Anjana Shenoy, Cindy Church

Technical Contributors and Reviewers

Debra Masada, Ajay Bharadwaj, Aurelio Garcia-Ribeyro, Steve Watt, Nancy Hildebrandt, Alla Redko, Tom McGinn, Matt Heimer, Sharon Zahkour, Nicolas Lorain, Brian Goetz, Joe Darcy, Alessandro Leite, Jai Suri, Richard Bair, Jasper Potts

This book was published using: Oracle Tutor

Table of Contents

Practices for Lesson 1: Introduction to the course

Chapter 1

Practices for Lesson 1

Practices Overview

There are no practices for this lesson.

Practices for Lesson 2: The BrokerTool Application

Chapter 2

Practices for Lesson 2

Practices Overview

In these practices, you set up your development environment and create the BrokerTool and HenleyApp databases. Once your environment is set up, you explore the BrokerTool and Ensemble applications. As an optional exercise, you can explore the HenleyApp application if time permits.

Practice 2-1: Setting Up Your Development Environment

Overview

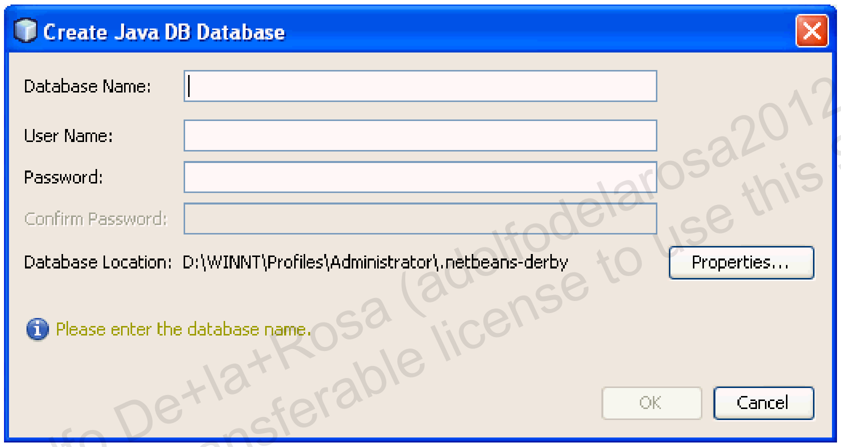
In this practice, you set up your development environment and create the BrokerTool database.

Assumptions

You have listened to the lecture and are prepared for the practice.

Task: Add the BrokerTool Database to Java DB

1. Start NetBeans by clicking the icon on the desktop.
2. Go to the Services tab, or from the menu, select Windows > Services.
3. Expand the Databases folder.
4. Right-click Java DB and choose Start Server.
5. Right-click Java DB and choose Create Database.



Create Java DB Database

Database Name:

User Name:

Password:

Confirm Password:

Database Location: D:\WINNT\Profiles\Administrator\.netbeans-derby

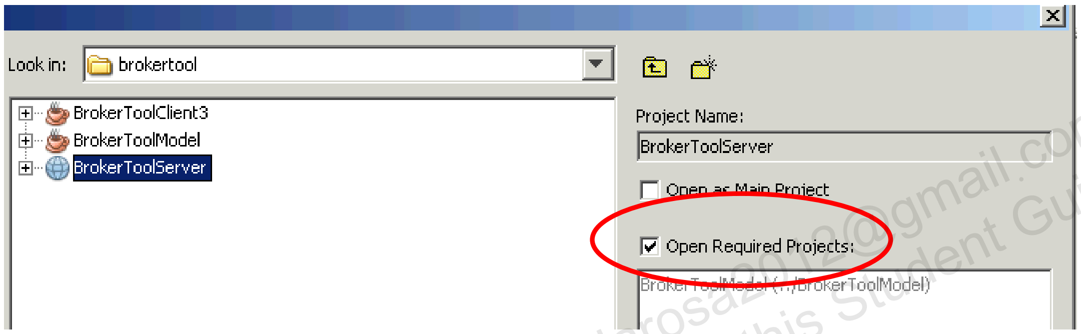
Please enter the database name.

6. Enter the following data in the form:
   - Database Name: `brokertool`
   - User Name: `brokertool`
   - Password: `brokertool`
   - Confirm Password: `brokertool`
7. Click OK and right-click and choose Connect to connect to the database.
8. Expand the Connection created for BrokerTool.
9. Expand the BrokerTool Database.
10. Right-click the Table folder and choose Execute Command.
11. Copy the contents of the `BrokerTool.sql` file located in the `D:\labs\resources\setup` directory into the SQL Command window.
12. Click the green arrow to run the SQL command that creates the database tables.
13. Right-click one of the tables and choose View Data to verify that the data was created.

Task: Install the BrokerTool Server

With the database installed, you need to install and deploy the BrokerTool web server. To deploy the server, perform the following steps:

1. If it is not started, start NetBeans.
2. Click the Services tab, or from the menu, select Windows > Services.
3. Expand the Servers folder.
4. Right-click the GlassFish server and choose Start.
5. Click the Projects tab, or from the menu, select Windows > Projects.
6. Open the BrokerToolServer project in the `D:\labs\demo\brokertool` directory.
7. Make sure you select the Open Required Projects check box.



Project Name: BrokerToolServer

Open as Main Project

[✓] Open Required Projects:

BrokerToolModel (../BrokerToolModel)

8. Look on the Projects tab and make sure that the end result is that both the BrokerToolServer project and the BrokerToolModel project are open.
9. Right-click BrokerToolModel and choose Build.
10. Build the BrokerToolServer.
11. Deploy the BrokerToolServer to GlassFish.

Task: Set Up GlassFish Authentication Users

You set up GlassFish user accounts for the authentication practices later in the course. You create two accounts, *Users* and *Group*, with the following settings:

- *Users* Username/Password
  - `user/user`
- *Group* Username/Password
  - `admin/admin`

To add these accounts, perform the following steps:

1. If it is not started, start NetBeans.
2. Click the Services tab, or from the menu, select Windows > Services.
3. Expand the Servers folder.
4. Right-click the GlassFish server and choose Start.
5. Open a Command Prompt window by choosing Start > Programs > Accessories > Command Prompt.

6. Change to the `GlassFish\bin` directory.

```
cd D:\Program Files\GlassFish-3.1.2\bin
```

7. Add the `user` account. Type the following at the command prompt:

```
asadmin create-file-user --groups Users user
```

8. You are prompted for a password. Set the password to `user`.
9. Add the `admin` account. Type the following at the command prompt:

```
asadmin create-file-user --groups Users admin
```

10. You are prompted for a password. Set the password to `admin`.
11. To verify that your users have been added, type:

```
asadmin list-file-users
```

12. If you make a mistake and need to remove a user, type:

```
asadmin delete-file-user username
```

Practice 2-2: Exploring the BrokerTool Application

Overview

In this practice, you set up the BrokerTool application and explore the code.

Assumptions

You have completed Practice 2-1 and your environment is set up.

Tasks

1. Open the JavaFX API documentation located in `D:\Program Files\Oracle\JavaFX 2.1 SDK\docs\api\index.html`. You can use the documentation for reference as you complete the practice.
2. From the NetBeans File menu, choose Open Project.
3. Navigate to `D:\labs\demo\brokertool\`.
4. While pressing the Shift key, select BrokerToolClient3, BrokerToolModel, and BrokerToolServer. Click Open Project.
5. In the Projects pane, right-click BrokerToolServer and choose Run.

   When the server is running, it opens a browser window and displays the message `BrokerToolServer is Running`.
6. Right-click BrokerToolClient3 and choose Run.
7. Log in with username `user` and password `user`.
8. Click the Create Order button and create a new order.
9. Explore the project files and code. If you have time, try modifying the code and running the application to see the changes that you made.

Practice 2-3: Running the Ensemble Sample Application

Overview

In this practice, you run the Ensemble sample application. Ensemble is a sample provided with every JavaFX release, and it contains samples of each type of JavaFX feature. It is a good tool to use when you are learning JavaFX.

Assumptions

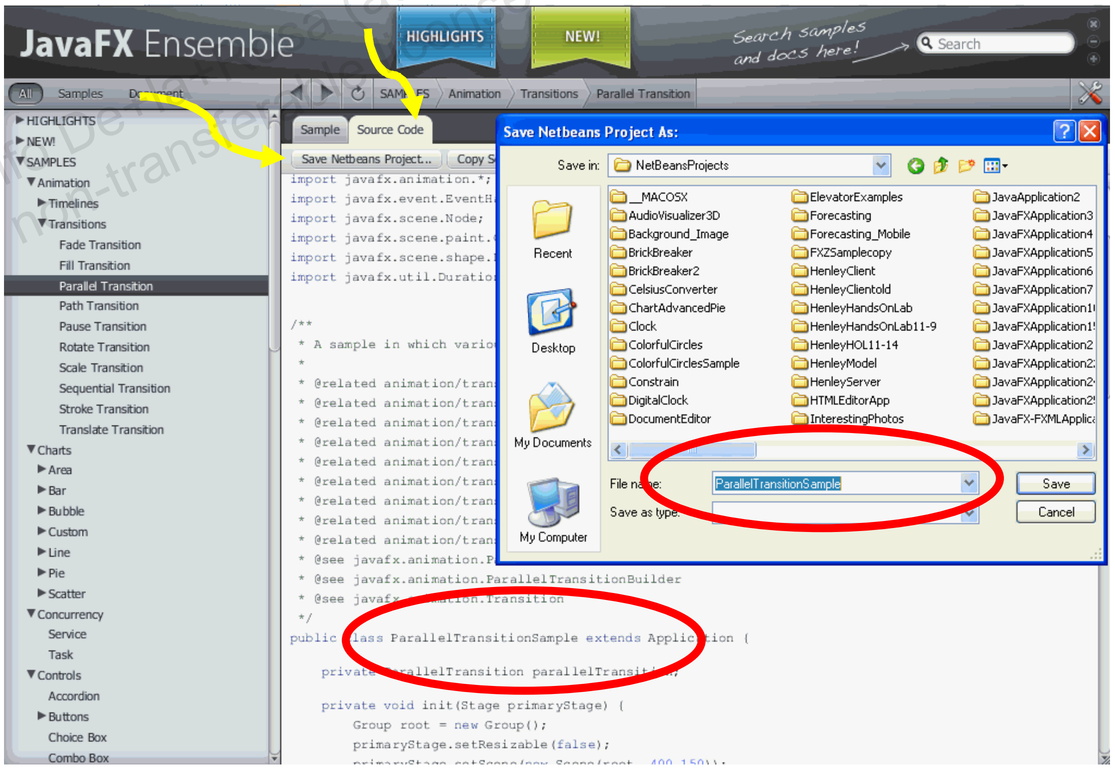
You have completed Practices 2-1 and 2-2.

Tasks

1. In NetBeans, choose File > Open Project.
2. Navigate to `D:\labs\demo\Ensemble`.
3. Click Open Project.
4. Explore the application.

Save a NetBeans Project from Ensemble

1. In Ensemble, choose an example that you would like to save.
2. Click the Source tab.
3. Click the Save NetBeans project button.
4. Give the NB project the same name as the example class name.



```
import javafx.animation.*;
import javafx.event.EventH
import javafx.scene.Node;
import javafx.scene.paint.
import javafx.scene.shape.
import javafx.util.Duratio


/**
 * A sample in which vario
 *
 * @related animation/tran
 * @related animation/tran
 * @related animation/tran
 * @related animation/tran
 * @related animation/tran
 * @related animation/tran
 * @related animation/tran
 * @related animation/tran
 * @related animation/tran
 * @see javafx.animation.P
 * @see javafx.animation.ParallelTransitionBuilder
 * @see javafx.animation.Transition
 */
public class ParallelTransitionSample extends Application {

    private ParallelTransition parallelTransition;

    private void init(Stage primaryStage) {
        Group root = new Group();
        primaryStage.setResizable(false);
```

Save in: NetBeansProjects

__MACOSX	ElevatorExamples	JavaApplication2
AudioVisualizer3D	Forecasting	JavaFXApplication3
Background_Image	Forecasting_Mobile	JavaFXApplication4
BrickBreaker	FXZSamplecopy	JavaFXApplication5
BrickBreaker2	HenleyClient	JavaFXApplication6
CelsiusConverter	HenleyClientold	JavaFXApplication7
ChartAdvancedPie	HenleyHandsOnLab	JavaFXApplication1
Clock	HenleyHandsOnLab11-9	JavaFXApplication1
ColorfulCircles	HenleyHOL11-14	JavaFXApplication2
ColorfulCirclesSample	HenleyModel	JavaFXApplication2
Constrain	HenleyServer	JavaFXApplication2
DigitalClock	HTMLEditorApp	JavaFXApplication2
DocumentEditor	InterestingPhotos	JavaFX-FXMLApplic

File name: ParallelTransitionSample

Save as type:

Practice 2-4: (Optional) Running the Henley Application

Overview

In this optional practice, you set up and run the Henley application. This application is used for demonstration purposes and is not part of the regular course exercises. It is provided to show an example of a more complex rich-client application.

Assumptions

You have completed Practices 2-1, 2-2, and 2-3.

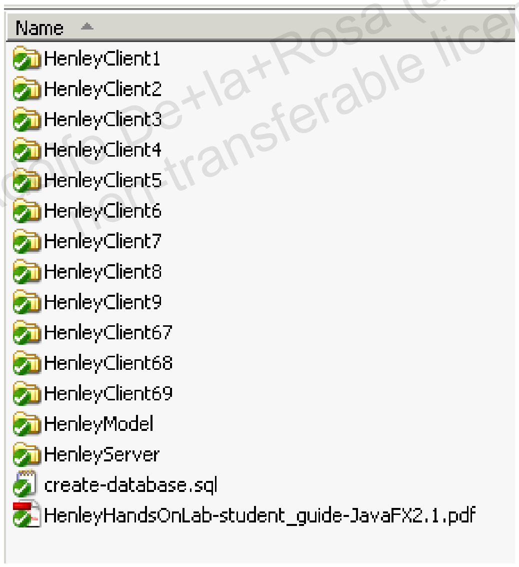
Task: Create the Henley Database

1. Follow the instructions in Practice 2-1 to create the Henley database. The name and settings for the database are:
   - Database Name: `henley`
   - User Name: `henley`
   - Password: `henley`
   - Confirm Password: `henley`

The SQL file is located in `D:\labs\demo\Henley\create-database.sql`.

Task: Review the Henley Projects

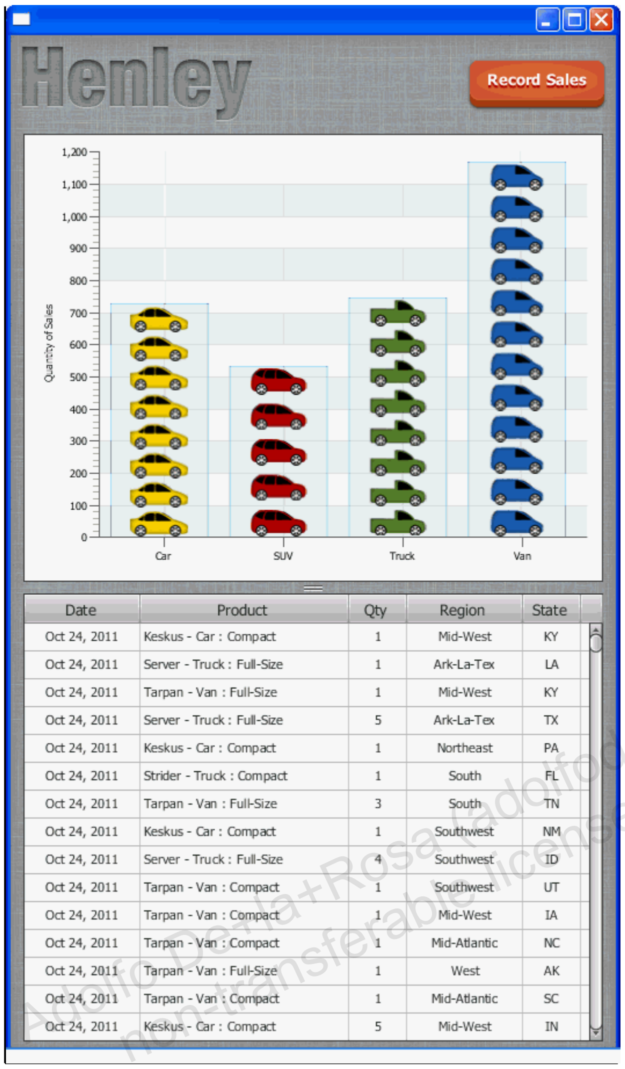
The `D:\labs\demo\Henley` folder contains several project files and a separate student guide that describes how to run each project.



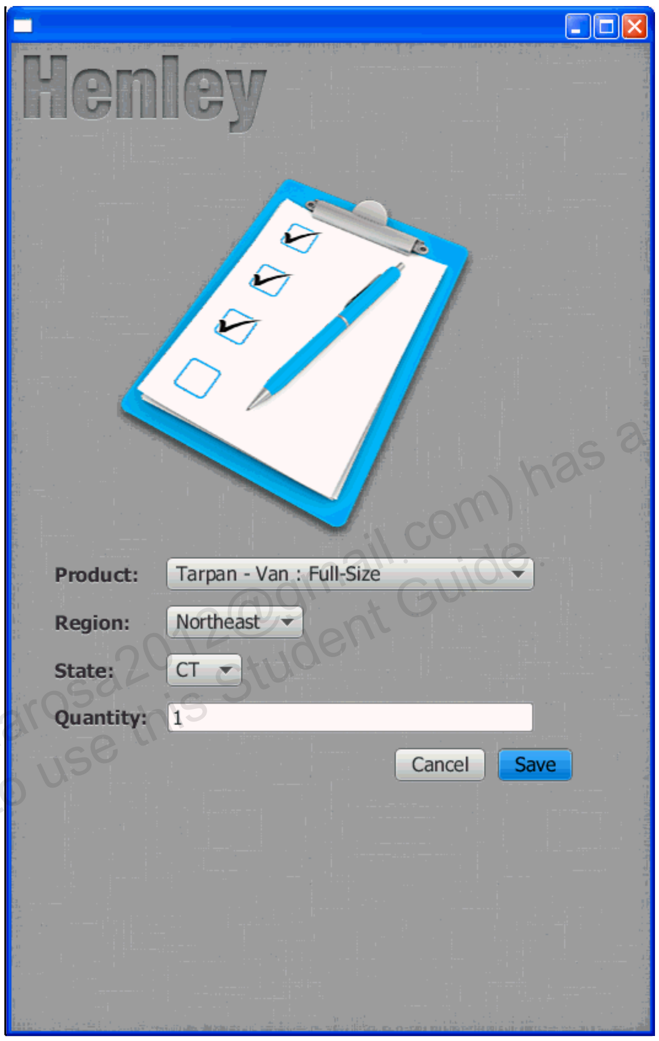
Each HenleyClient project builds on the previous client. Like the BrokerTool application, the Henley application uses server and model projects that need to be open and deployed before you can use the Henley application.

1. In NetBeans, open HenleyClient9, HenleyModel, and HenleyServer.
2. Right-click HenleyServer and choose Run.

3. Right-click HenleyClient9 and choose Run. The main window opens. If you click the Record Sales button, you can see the Order Form window.



Date	Product	Qty	Region	State
Oct 24, 2011	Keskus - Car : Compact	1	Mid-West	KY
Oct 24, 2011	Server - Truck : Full-Size	1	Ark-La-Tex	LA
Oct 24, 2011	Tarpan - Van : Full-Size	1	Mid-West	KY
Oct 24, 2011	Server - Truck : Full-Size	5	Ark-La-Tex	TX
Oct 24, 2011	Keskus - Car : Compact	1	Northeast	PA
Oct 24, 2011	Strider - Truck : Compact	1	South	FL
Oct 24, 2011	Tarpan - Van : Full-Size	3	South	TN
Oct 24, 2011	Keskus - Car : Compact	1	Southwest	NM
Oct 24, 2011	Server - Truck : Full-Size	4	Southwest	ID
Oct 24, 2011	Tarpan - Van : Compact	1	Southwest	UT
Oct 24, 2011	Tarpan - Van : Compact	1	Mid-West	IA
Oct 24, 2011	Tarpan - Van : Compact	1	Mid-Atlantic	NC
Oct 24, 2011	Tarpan - Van : Full-Size	1	West	AK
Oct 24, 2011	Tarpan - Van : Compact	1	Mid-Atlantic	SC
Oct 24, 2011	Keskus - Car : Compact	5	Mid-West	IN



Product: Tarpan - Van : Full-Size
Region: Northeast
State: CT
Quantity: 1

4. Open the student guide `HenleyHandsOnLab-student_guide-JavaFX2.1.pdf` and follow the instructions to complete various lessons. You can work on this project throughout the five days when you have time.

Practices for Lesson 3: JavaFX

Chapter 3

Practices for Lesson 3

Practices Overview

In these practices, you create JavaFX and FXML projects and build a login screen UI that is used in the final BrokerTool application.

Practice 3-1: (Summary Level) Creating a Login Window by Using JavaFX

Overview

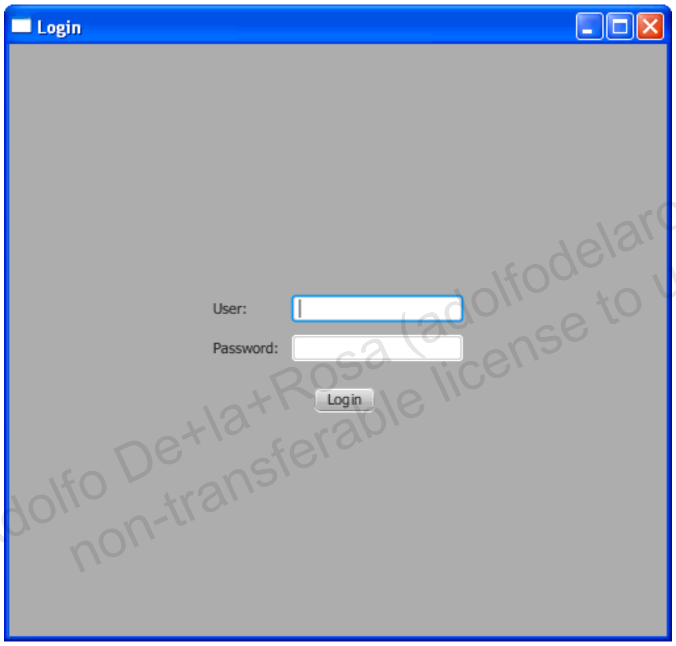
In this practice, you create a JavaFX application using the NetBeans IDE. You explore some of the JavaFX features and become familiar with the development environment.

Assumptions

You have listened to the lecture and are familiar with the content.

Summary

Create a Login screen that has a text field for the user name, a password field for the password, and a button to log in. This task introduces JavaFX components such as a TextField, PasswordField, and the GridPane layout manager. The Login screen should look like the following:



Task: Creating a New JavaFX Application

1. In NetBeans, create a new JavaFX application named Login, and the application class name is `com.example.login.gui.Login`.
2. Delete the default code after the `main()` method.
3. Use the start() method to build a stage and scene with the title Login, the sizes 520 by 450, and the color darkgrey.
4. Right-click in the editor and choose Fix Imports. Be sure to select the `javafx` package names and not the `java.awt` package names.
5. Declare an instance of a GridPane, which will be the root node of the scene. Place the node below the `main()` method.

6. Use the `init()` method to specify the GridPane and the contents of the scene. Add the code below the stage block of code.
7. Use the following parameters for the GridPane:

Parameter	Value
`Id`	`rootPane`
`Vgap`, `Hgap`	10
`Padding`	10 on top, right, bottom, left sides
`Alignment`	`center`

8. Add two Labels, a TextField, and a PassWord field to the GridPane using the following parameters:

Parameter	Variable name	Value
Label name	`userLabel`	`User:`
Label position		column 1 row 2
TextField name	`userTextField`	
TextField position		column 2 row 2
Label name	`pwdLabel`	`Password:`
Label position		column 1 row 3
PasswordField	`pwdField`	
PasswordField position		column 2 row 3

9. Create an HBox as a child element of the GridPane that spans three columns and is in column 1 and row 5 of the GridPane.
10. Add a button named `Login` and align it in the center of the HBox.

    Note: You do not add an action to the button at this time. Event handling is covered in a later lesson.
11. Right-click and choose Fix Imports.
12. Run the application. The application should look like the image at the beginning of this section.

Practice 3-1: (Detail Level) Creating a Login Window by Using JavaFX

Overview

In this practice, you create a JavaFX application using the NetBeans IDE. You explore some of the JavaFX features and become familiar with the development environment.

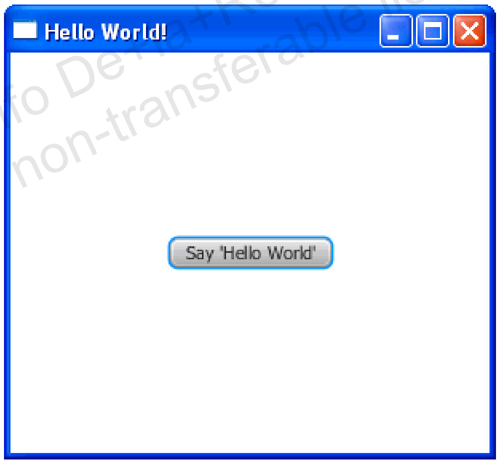
Assumptions

You have listened to the lecture and are familiar with the content.

Task: Running a Default JavaFX Application.

1. Open the JavaFX API documentation located in `D:\Program` **`Files\Oracle\JavaFX 2.1 SDK\docs\api\index.html`**. You can use the documentation for reference as you complete the practice.
2. From the NetBeans File menu, choose New Project.
3. In the New Project dialog box, select JavaFX under the Categories column, and choose JavaFX Application. Click Next.
    a. Name the project Login
    b. Name the application class `com.example.login.gui.Login`.
    c. Save the file in `D:\labs\lab03-IntroJavaFX\practices\practice01`.
    d. Click Finish.

    NetBeans creates a small JavaFX application by default.
4. Run the project and see the results.



Task: Create the Stage and Scene.

1. Open `Login.java`.
2. Delete the code below the `main()` method class declaration in `Login.java`.

```
    public static void main(String[] args) {
        launch(args);
    }
//delete code below here
```

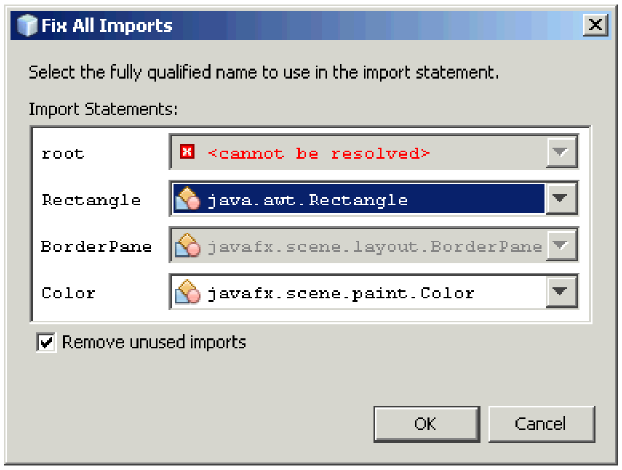
3. Use the `start()` method to build a stage and scene with the title `Login`, the sizes 520 by 450, and the color `darkgrey`.

   The `start()` method is the main entry point for all JavaFX applications. The `start()` method is called after the `init()` method has returned and after the system is ready for the application to begin running. This method is called on the JavaFX Application thread.

```
    public class Login extends Application {

      public static void main(String[] args) {
          launch(args);
      }
  //add code here
  @Override
      public void start(Stage primaryStage) {
          primaryStage.setTitle("Login");
          primaryStage.setScene(new Scene(root, 500, 450,
                                 Color.DARKGREY));
          primaryStage.show();
   }
  }
```

4. Right-click in the editor and choose Fix Imports.

   In the Fix Imports dialog box, you see that you have the choice of either `java.awt` packages or `javafx` packages.



Select the fully qualified name to use in the import statement.

Import Statements:

root	<cannot be resolved>
Rectangle	java.awt.Rectangle
BorderPane	javafx.scene.layout.BorderPane
Color	javafx.scene.paint.Color

☑ Remove unused imports

5. Choose the `javafx` package options to fix the imports.

   Note: You cannot run the application at this point because you have not declared a root node.

Task: Create the GridPane and Components

In this task, you create a GridPane and set the Id, padding, and alignment.

The GridPane is a layout manager. Nodes can be placed in any cell in the grid and can span cells as needed. A grid pane is useful for creating forms or any layout that is organized in rows and columns. Lesson 5 includes a detailed description of a GridPane.

1. Declare an instance of a GridPane, which will be the root node of the scene. Place the node below the `main()` method. Your code should look similar to the following:

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

GridPane root = new GridPane();
```

2. Use the `init()` method to specify the GridPane and the contents of the scene. Add the code below the stage block of code.

   The `init()` method is the application initialization method. This method is not called on the JavaFX Application thread. An application must not construct a scene or a stage in this method. An application may construct other JavaFX objects in this method.

   The GridPane should use the following parameters:

Parameter	Value
`Id`	`rootPane`
`Vgap, Hgap`	10
`Padding`	10 on top, right, bottom, left sides
`Alignment`	`center`

   Note: GridPane lays out its children within a flexible grid of rows and columns. If a border and/or padding is set, its content will be laid out within those insets. You can choose to set the GridLines to visible to see the sections of the grid. Note that the following code sample includes a commented line that shows how to set the GridLines.

Your code should look similar to the following:

```
@Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Login");
        primaryStage.setScene(new Scene(root, 500, 450,
                                  Color.DARKGREY));
        primaryStage.show();
}
//add GridPane here
@Override
    public void init() {
        root.setId("rootPane");
        root.setVgap(10);
        root.setHgap(10);
        root.setPadding(new Insets(10, 10, 10, 10));
        root.setAlignment(Pos.CENTER);
       //root.setGridLinesVisible(true);
```

The login window uses two Label and two TextField components to create the interface.

3. Create an instance of a Label with the title `User:` in column 1 row 2 of the GridPane.

```
@Override
    public void init() {
        root.setId("rootPane");
        root.setVgap(10);
        root.setHgap(10);
        root.setPadding(new Insets(10, 10, 10, 10));
        root.setAlignment(Pos.CENTER);

        Label userLabel = new Label("User:");
        root.add(userLabel, 1, 2);
```

4. Create an instance of a TextField that is the field for the username entry in column 2 row 2.

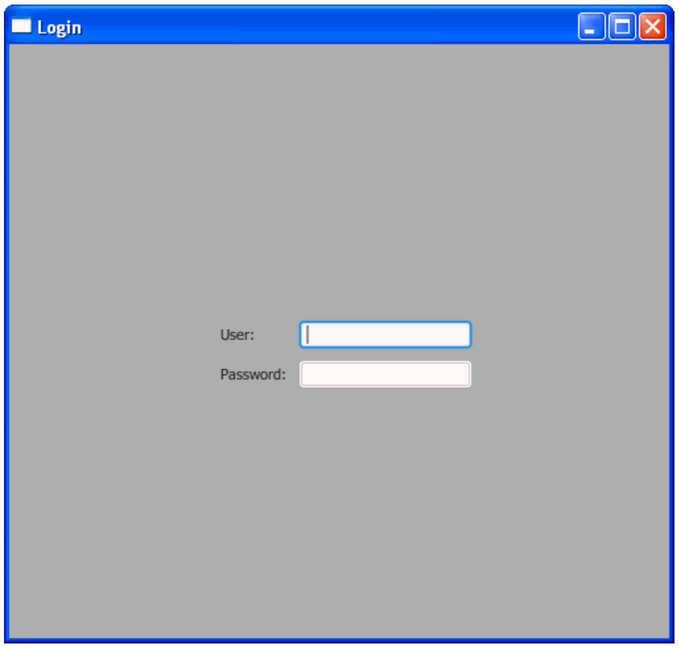
```
TextField userTextField = new TextField();
root.add(userTextField, 2, 2);
```

5. Add the Label and PasswordField for the password controls using the following values:

Parameter	Variable name	Value
Label name	`pwdLabel`	`Password:`
Label position		column 1 row 3
PasswordField	`pwdField`	
PasswordField position		column 2 row 3

6. Right-click and choose Fix Imports.

7. Run the application. The output should look like the following:



Does your application look different from the image? Here is what your code should look like up to this point:

```
public class Login extends Application {

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

    GridPane root = new GridPane();

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Login");
        primaryStage.setScene(new Scene(root, 500, 450,
                                         Color.DARKGREY));
        primaryStage.show();
    }

    @Override
    public void init() {
        root.setId("rootPane");
        root.setVgap(10);
        root.setHgap(10);
        root.setPadding(new Insets(10, 10, 10, 10));
        root.setAlignment(Pos.CENTER);
        //root.setGridLinesVisible(true);

        Label userLabel = new Label("User:");
        root.add(userLabel, 1, 2);
        TextField userTextField = new TextField();
        root.add(userTextField, 2, 2);
        Label pwdLabel = new Label("Password:");
        root.add(pwdLabel, 1, 3);
        PasswordField pwdField = new PasswordField();
        root.add(pwdField, 2, 3);
    }
}
```

Task: Build an HBox with a Button

In this task, you create an HBox as a child element of the GridPane. The Login button is positioned in the HBox container.

1. Create an HBox that spans three columns and is in column 1 and row 5 of the GridPane.

   For a hint on positioning the HBox, look at the GridPane class in the API documentation. You can set the position of the box by using integers with the following constraints: columnIndex, rowIndex, columnSpan, rowSpan.

   Your code should look like the following:

```
        //Create the labels and textfields.
        Label userLabel = new Label("User:");
        root.add(userLabel, 1, 2);
        TextField userTextField = new TextField();
        root.add(userTextField, 2, 2);
        Label pwdLabel = new Label("Password:");
        root.add(pwdLabel, 1, 3);
        PasswordField pwdField = new PasswordField();
        root.add(pwdField, 2, 3);
        //add HBox here
       HBox buttonBox = new HBox();
       root.add(buttonBox, 1, 5, 3, 1);
```

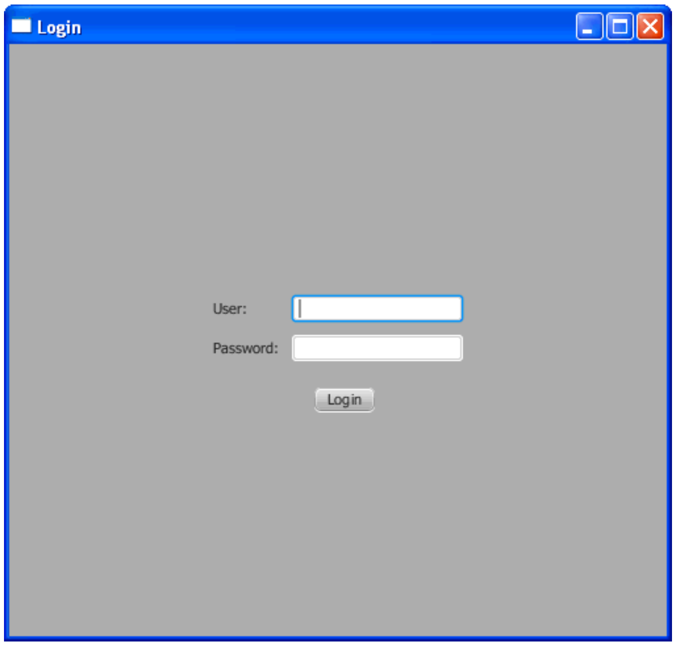
2. Add a button named `Login` and align it in the center of the HBox.

```
Button loginBtn = new Button("Login");
buttonBox.setAlignment(Pos.CENTER);
buttonBox.getChildren().add(loginBtn);
}
```

   Note: You do not add an action to the button at this time. Event handling is covered in a later lesson.

3. Run the project.

The finished project should like the following image:



You can compare your results with the solution file in `D:\labs\lab03-IntroJavaFX\solutions\solution01`.

Practice 3-2: (Summary Level) Creating an FXML Login Window

Overview

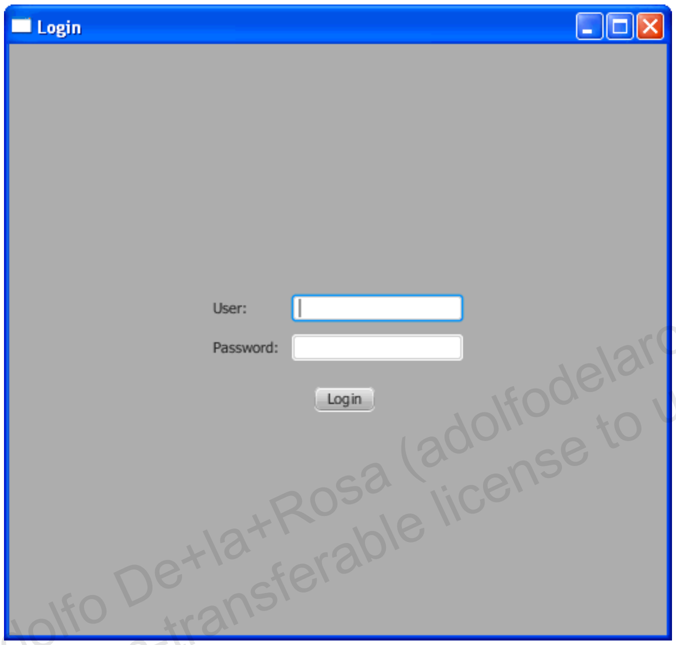
In this practice, you create the same login screen as the previous practice, except you develop the screen using FXML.

Assumptions

You have completed the first practice.

Summary

You create a Login window that looks similar to the following image:



Task: Create an FXML Application

1. In NetBeans, choose File > New Project > JavaFX > JavaFX FXML Application.
2. Name the application LoginFXML and name the application class `com.example.login.gui.LoginFXML`
3. In the projects pane, right-click a file name and choose Refactor > Rename. (The FXML file has the Rename option only.) Use the following values:

Original Name	New Name	Purpose
`LoginFXML.java`	`LoginFXMLApp.java`	Launches the application
`Sample.FXML`	`LoginFXML.FXML`	Application interface
`Sample.java`	`LoginFXML.java`	Controller

4. Open `LoginFXMLApp.java` and change the resource file name from `Sample.fxml` to `LoginFXML.fxml`.
5. Right-click the LoginFXML project name and choose Properties > Run.

6. Change the Application Class name to `com.example.login.gui.LoginFXMLApp` and click OK.
7. Open `LoginFXML.java` and remove the Label variable and Button event handling.
8. Open `LoginFXMLApp.java` and size the scene as 500 and 450 and set its color as `darkgrey`.
9. Right-click and choose Fix Imports.
10. Open `LoginFXML.fxml` and delete all of the AnchorPane code.
11. Create an instance of a GridPane that uses the following values:

Parameter	Value
`xmlns:fx`	`http://javafx.com/fxml`
`fxcontroller`	`com.example.login.gui.LoginFXML`
`fx:id`	`login`
`hgap`	`15`
`vgap`	`12`
`alignment`	`center`

12. Add padding to the GridPane using the value `10` for the top, right, bottom, and left sides.
13. Add the import statement for the `Insets` class.

```
<?import javafx.geometry.Insets?>
```

14. Create an instance of a Label in `LoginFXML.fxml`. The Label is a child element of GridPane. The `fx:id` is `User:` and index is column 1 row 2.
15. Add three more controls below the `User:` label using the following values:

Control Type	Parameter	Value
TextField	`fx:id`	`user`
	index	column 2 row 2
Label	`text`	`Password:`
	index	column 1 row 3
PasswordField	`fx:id`	`password`
	index	column 2 row 3

16. Add an HBox that is the container for the login button. The HBox is a child element of the GridPane and uses the following values:

Parameter	Value
`spacing`	`10`
`alignment`	`center`
index	column 1 row 5 span 3

17. Add a Button that is a child element of HBox, and name the button `Login`.
18. Run the project.

Practice 3-2: (Detail Level) Creating an FXML Login Window

Overview

In this practice, you create the same login screen as the previous practice, except you develop the screen using FXML.

Assumptions

You have completed the first practice.

Task: Running a Default FXML Application

1. In your browser, choose File > Open File and navigate to `D:\labs\03_IntroJavaFX\resources\introduction_to_fxml.html`.
2. Click Open.

   You can use this reference as you work through the FXML practice.
3. From the NetBeans File menu, choose New Project.
4. In the New Project dialog box, select JavaFX in the Categories column and choose JavaFX FXML. Click Next.
   a. Name the project LoginFXML.
   b. Name the application class com.example.login.gui.LoginFXML.
   c. Save the file in D:\labs\lab03-IntroJavaFX\practices\practice02.
   d. Click Finish.
5. Right-click and run the project. You see that it is similar to the default application that you created in Practice 1.

   NetBeans created three default files when you built the FXML project. You can rename the files to names that are specific for your application.
6. In the projects pane, right-click a file name and choose Refactor > Rename. (The FXML file has the Rename option only.) Use the following values:

Original Name	New Name	Purpose
`LoginFXML.java`	`LoginFXMLApp.java`	Launches the application
`Sample.FXML`	`LoginFXML.FXML`	Application interface
`Sample.java`	`LoginFXML.java`	Controller

7. Open `LoginFXMLApp.java`.
8. Notice that the following line of code is unique to FXML.

```
Parent root =
FXMLLoader.load(getClass().getResource("Sample.fxml"));
```

   The `FXMLLoader.load()` method loads the object hierarchy from the resource file `Sample.fxml` and assigns it to the variable named root. Because you changed the name of the resource file in your application, you need to modify the code.
9. Change the resource file name from `Sample.fxml` to `LoginFXML.fxml`.

10. Right-click the LoginFXML project name and choose Properties > Run.
11. Change the Application Class name to `com.example.login.gui.LoginFXMLApp` and click OK.
12. Open `LoginFXML.java` and remove the Label variable and Button event handling. You do not need that code for this practice. The code should look like the following after you make the changes:

```
public class LoginFXML implements Initializable {

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        // TODO
    }
}
```

In the next Task, you create the scene and GridPane.

Task: Create the Scene and GridPane

1. Open `LoginFXMLApp.java` and size the scene as 500 and 450 and color `darkgrey`.

```
stage.setScene(new Scene(root, 500, 450, Color.DARKGREY));
```

2. Right-click and choose Fix Imports.
3. Open `LoginFXML.fxml` and delete all of the AnchorPane code.
4. Create an instance of a GridPane that uses the following values:

Parameter	Value
`xmlns:fx`	`http://javafx.com/fxml`
`fxcontroller`	`com.example.login.gui.LoginFXML`
`fx:id`	`login`
`hgap`	15
`vgap`	12
`alignment`	`center`

The format looks like this:

```
<GridPane xmlns:fx="http://javafx.com/fxml" fx:controller="foo"
fx:id="foo" … >
</GridPane>
```

5. Add padding to the GridPane using the value `10` for the top, right, bottom, and left sides. The insets use the following tags:

```
<padding><Insets … ></padding>
```

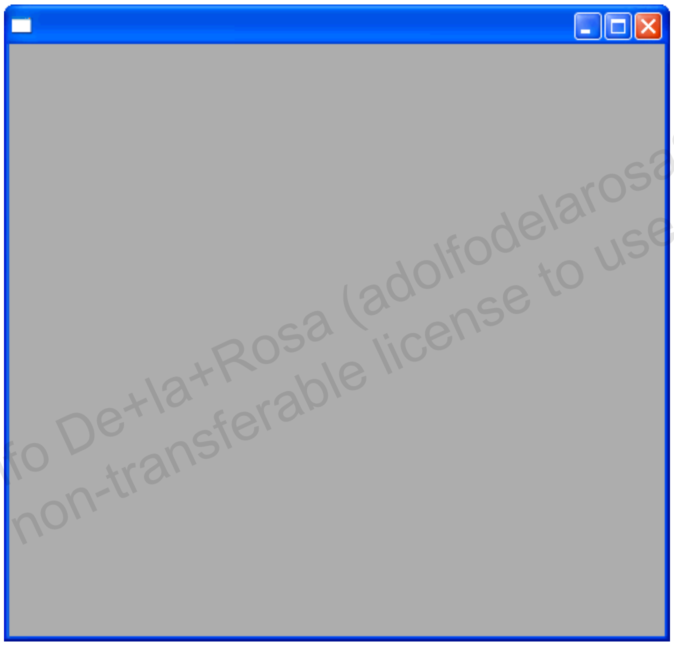
6. Add the import statement for the `Insets` class.

```
<?import javafx.geometry.Insets?>
```

Your code up to this point should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.Insets?>

<GridPane xmlns:fx="http://javafx.com/fxml"
fx:controller="com.example.login.gui.LoginFXML" fx:id="login"
hgap="15" vgap="12" alignment="center">
    <padding><Insets top="10" right="10" bottom="10"
left="10"/></padding>
</GridPane>
```

7. Run the project.



At this point in the project, you see an empty stage. In the next task, you create the login controls.

Task: Add Controls to the Login Window

1. Create an instance of a Label in `LoginFXML.fxml`. The Label is a child element of GridPane. The `fx:id` is `User:` and index is column 1 row 2. The code uses the following format:

```
<children>
        <Label text="User:" GridPane.columnIndex="1"
GridPane.rowIndex="2"/>
</children>
```

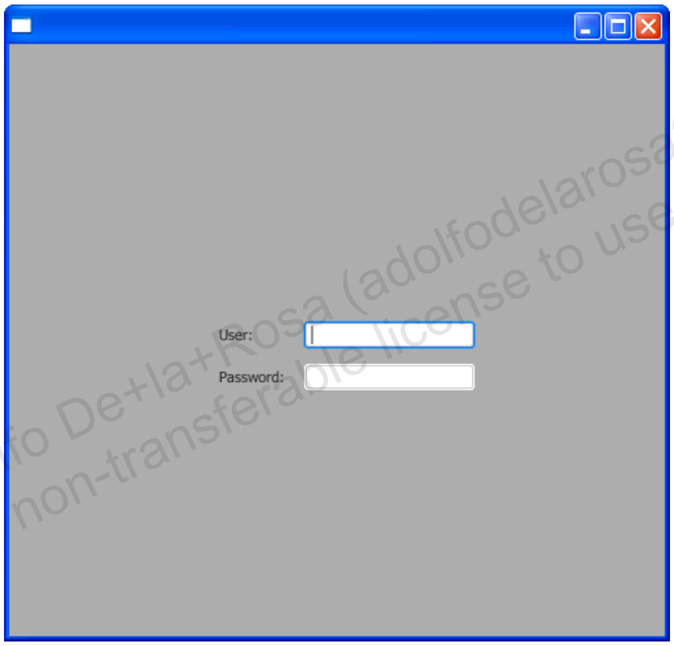
2. Add three more controls below the `User:` label using the following values:

Control Type	Parameter	Value
TextField	`fx:id`	`user`
	index	column 2 row 2
Label	`text`	`Password:`
	index	column 1 row 3
PasswordField	`fx:id`	`password`
	index	column 2 row 3

Note: You did not have to add the import statement `<?import javafx.scene.control.*?>` because it was in the code from the original default project that you created.

3. Run the project.

The output should look like the following:



Here is what your code should look like so far:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.Insets?>

<GridPane xmlns:fx="http://javafx.com/fxml"
fx:controller="com.example.login.gui.LoginFXML" fx:id="login"
hgap="15" vgap="12" alignment="center">
    <padding>
        <Insets top="10" right="10" bottom="10" left="10"/>
    </padding>
    <children>
        <Label text="User:" GridPane.columnIndex="1"
GridPane.rowIndex="2"/>
        <TextField fx:id="user"  GridPane.columnIndex="2"
GridPane.rowIndex="2"/>
        <Label text="Password:" GridPane.columnIndex="1"
GridPane.rowIndex="3"/>
        <PasswordField fx:id="password" GridPane.columnIndex="2"
GridPane.rowIndex="3"/>
    </children>
</GridPane>
```

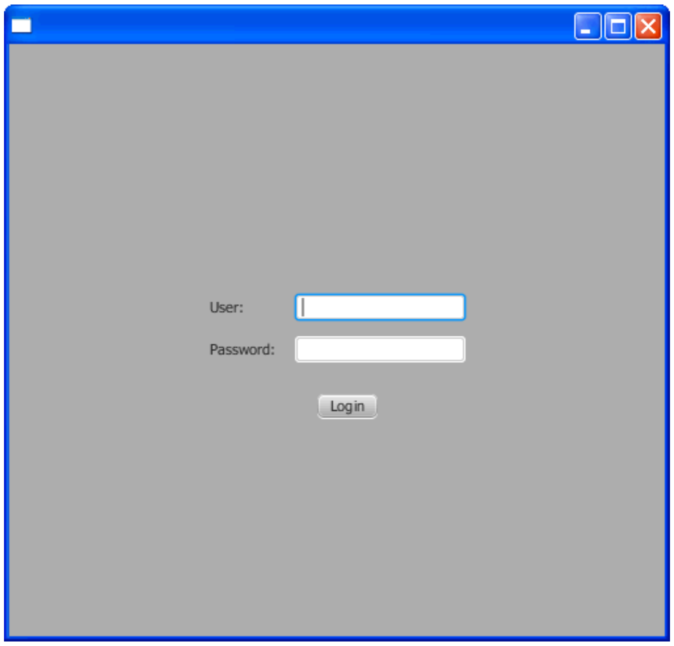
4. Add an HBox that is the container for the login button. The HBox is a child element of the GridPane. Use the following values:

Parameter	Value
`spacing`	`10`
`alignment`	`center`
index	column 1 row 5 span 3

5. Add a Button that is a child element of HBox, and name the button `Login`.

   Note: You do not add an action to the button at this time. Event handling is covered in a later lesson.

6. Run the project. It should look like the following:



The final code in LoginFXML.fxml should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>


<?import java.lang.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.Insets?>

<GridPane xmlns:fx="http://javafx.com/fxml"
fx:controller="com.example.login.gui.LoginFXML" fx:id="login"
hgap="15" vgap="12" alignment="center">
    <padding>
        <Insets top="10" right="10" bottom="10" left="10"/>
    </padding>
    <children>
        <Label text="User:" GridPane.columnIndex="1"
GridPane.rowIndex="2"/>
        <TextField fx:id="user"  GridPane.columnIndex="2"
GridPane.rowIndex="2"/>
        <Label text="Password:" GridPane.columnIndex="1"
GridPane.rowIndex="3"/>
        <PasswordField fx:id="password" GridPane.columnIndex="2"
GridPane.rowIndex="3"/>
```

```
        <HBox spacing="10" alignment="center"
GridPane.columnIndex="1" GridPane.rowIndex="5"
GridPane.columnSpan="3">
            <children>
                <Button text="Login"/>
            </children>
        </HBox>
    </children>
</GridPane>
```

You can compare your results with the solution file in `D:\labs\lab03-IntroJavaFX\solutions\solution02`.

Practices for Lesson 4: Generics and JavaFX Collections

Chapter 4

Practices for Lesson 4

Practices Overview

In these practices, you create `ObservableList` objects and create event listeners.

Practice 4-1: (Summary Level) Adding a Second Listener to Your List

Overview

In this practice, you add a second listener to the `ListExample` project.

Assumptions

You have listened to the lecture and are familiar with the content.

Tasks

1. Open the `ListListener` project in the `practice01` directory for this project.
2. Edit the `ListExample.java` file.
3. Add code to display the size of the lists.
4. Add code to add items to the list.
5. Build and run the project. Correct any errors.

Practice 4-1: (Detail Level) Adding a Second Listener to Your List

Overview

In this practice, you add a second listener to the `ListExample` project.

Assumptions

You have listened to the lecture and are familiar with the content.

Tasks

1. Open the `ListListener` project in the `practice01` directory for this project.
2. Edit the `ListExample.java` file.
3. Add code to display the size of the lists. For example, your event handler might look like this:

```
observableList.addListener(new ListChangeListener() {


  @Override
    public void onChanged(ListChangeListener.Change change) {
    System.out.println("Observable list length: " +
observableList.size() + "  List length: " + list.size());


   }
 });
```

4. Add code to add items to the list.

```
        observableList.add("John");
        observableList.add("Paul");
        observableList.add("George");
        observableList.add("Ringo");
```

5. Build and run the project. Correct any errors.

Practices for Lesson 5: UI Controls, Layouts, Charts, and CSS

Chapter 5

Practices for Lesson 5

In the practices for this lesson, you create various components for the BrokerTool application by using JavaFX.

Practice 5-1: (Summary Level) Creating an Order Form by Using FXML

Overview

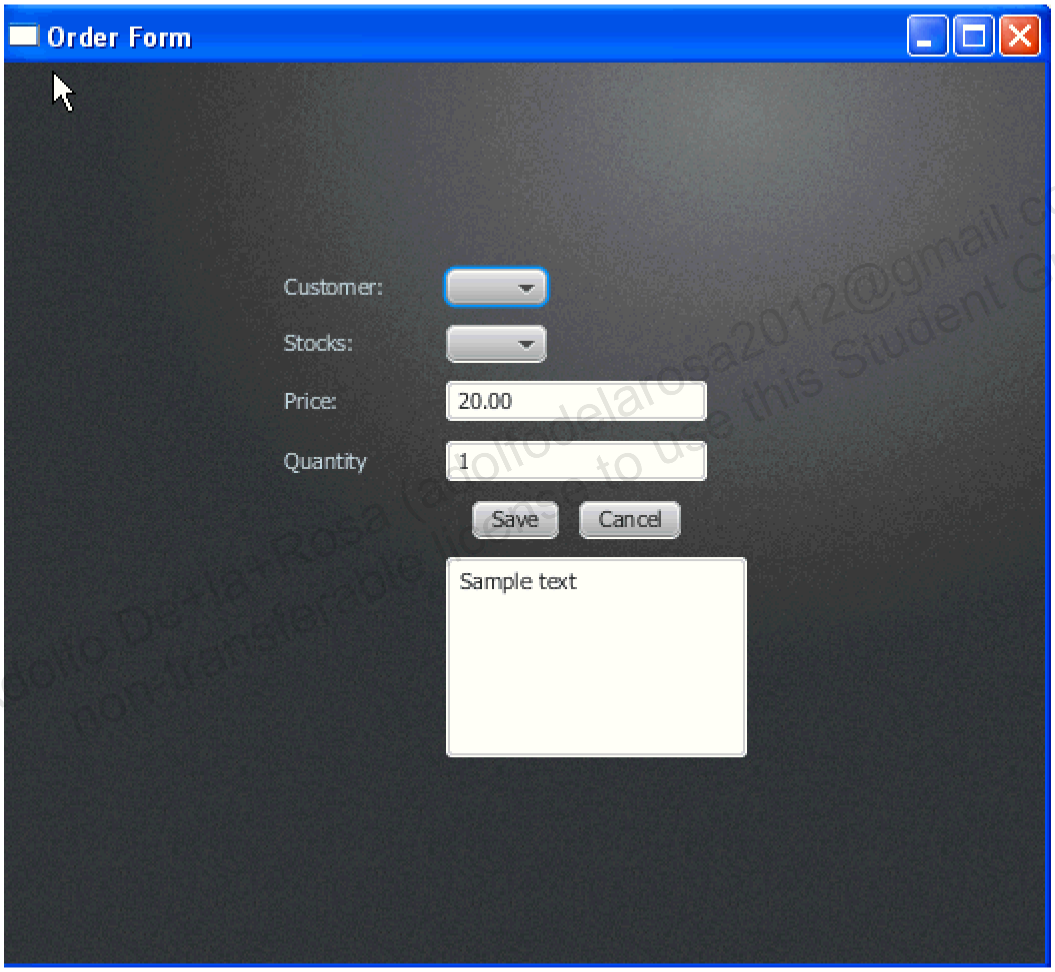
In this practice, you create an order form for the BrokerTool application by using FXML.

Assumptions

You have listened to the lecture portion of this lesson.

Summary

You will create a small application that contains all the interface controls for the stock order form. When done, your application should look like this:



Note that no events are yet associated with the application. Use the GridPane layout to achieve the result shown.

Tasks

1. Open the `OrderDialog` project from the student practice directory.
2. Make sure it is set as the Main Project in NetBeans.
3. Open and review `OrderDialog.java`.
4. Open the `OrderDialog.java` file. You will build your UI in this file.

5. Set up the GridPane to have five columns. Two columns will provide only 10 pixels of space.
6. Set the padding to 10, set the `hgap` and `vgap` to 10, set the ID to `rootPane`, and make the default alignment centered.
7. Add a `children` element to the GridPane. Add additional elements within this `children` element.
8. Add a `Label` element with the following characteristics to the first row of the GridPane.
   a. Set the ID to `customerLabel`.
   b. Set the text to `Customer:`
   c. Place it in column 0, row 0.
9. Add a `ChoiceBox` element with the following characteristics to the first row of the GridPane.
   a. Set the ID to `customerChoiceBox`.
   b. Place the `ChoiceBox` in column 2, row 0.
10. Add a `Label` with following characteristics to the second row of the GridPane.
    a. Set the ID to `stockLabel`.
    b. Set the text to `Stock:`
    c. Place it in column 0, row 1.
11. Add a `ChoiceBox` with the following characteristics to the second row of the GridPane.
    a. Set the ID to `stockChoiceBox`.
    b. Place it in column 2, row 1.
12. Add a `Label` with the following characteristics to the third row of the GridPane.
    a. Set the ID to `priceLabel`.
    b. Set the text to `Price:`
    c. Place it in column 0, row 2.
13. Add a `TextField` with the following characteristics to the third row of the GridPane.
    a. Set ID to `priceTextLabel`.
    b. Place it in column 0, row 2.
14. Add a `Label` element with the following characteristics to the fourth row of the GridPane.
    a. Set the ID to `qtyLabel`.
    b. Set the text to `Quantity:`
    c. Place it in column 0, row 3.

15. Add a `TextField` with the following characteristics to the fourth row of the GridPane
    a. Set the ID to `quantityTextField`.
    b. Place it in column 2, row 3.
16. Add an `HBox` to the fifth row of the GridePane.
    a. Set the ID of the HBox to `buttonHBox` and put the element in column 2, row 4. Have it span two columns.
    b. Add a `children` element.
    c. Add two buttons inside the `children` element.
17. Configure the resultsTextArea text area and add it row 6 of the GridPane. Have it span two columns
    a. Set the text to "Sample Text".
    b. Set the preferred width to 300 and the preferred height to 100.
18. Save and Run the application. Correct any errors.
19. Open the `OrderDialog.css` file. Note the styles that are defined.
    a. Notice the CSS selectors and the rules and how they are defined in JavaFX.
    b. For the `rootPane` note the background styling.
    c. Close the file.
20. Add the OrderDialog.css to the application. Use the following code in the location indicated in the comments.

```
scene.getStylesheets().add(OrderDialogApp.class.getResource("Ord
erDialog.css").toExternalForm());
```

21. Run the application and ensure that it looks like the image shown at the beginning of this practice.

Practice 5-1: (Detail Level) Creating an Order Form by Using FXML

Overview

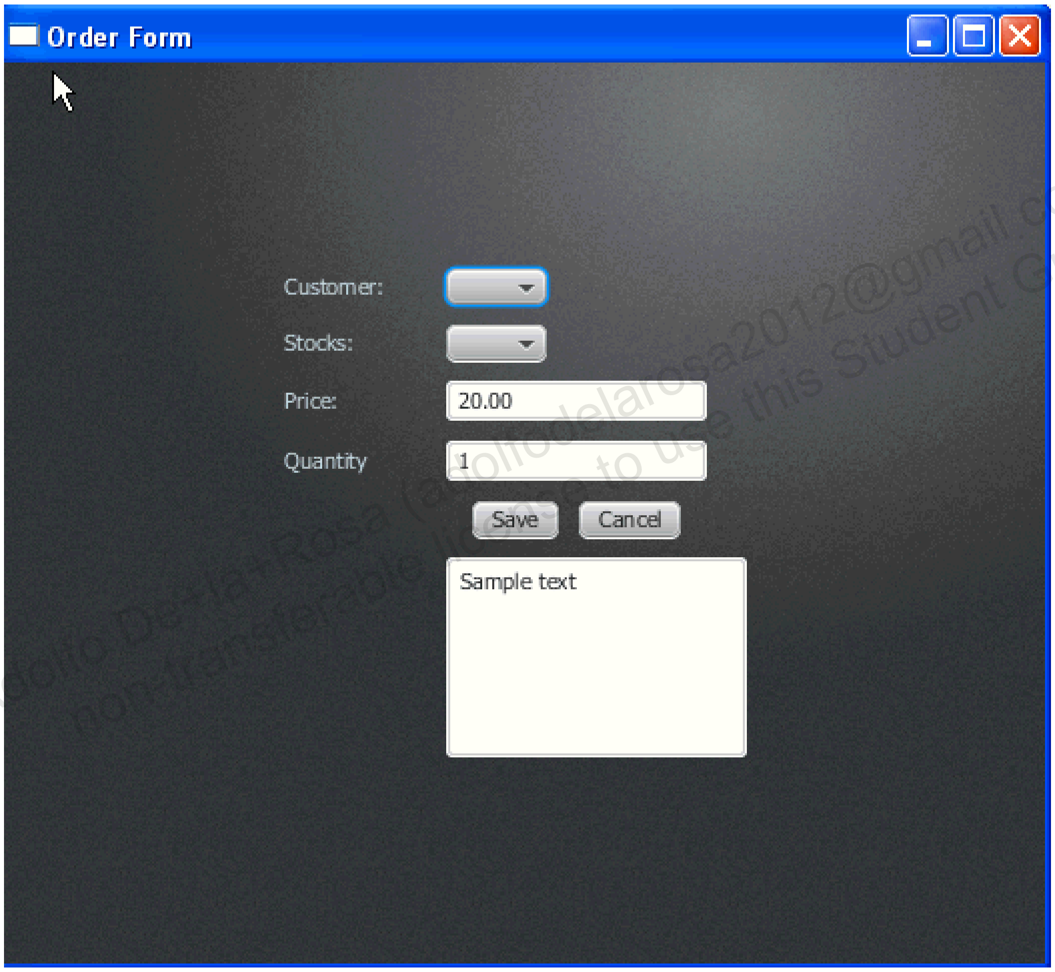
In the practice, you create an order form for the BrokerTool application by using FXML.

Assumptions

You have listened to the lecture portion of this lesson.

Summary

Create a small application that contains all the interface controls for the stock order form. When done, your application should look like this:



Note that no events are yet associated with the application. Use the GridPane layout to achieve the result shown.

Tasks

1. Open the `OrderDialog` project from the practice directory.
2. Make sure it is set as the Main Project in NetBeans.
3. Open and review `OrderDialog.java`.
4. Open the `OrderDialog.java` file. You will build your UI in this file.

5. Set up the GridPane to have five columns. Two columns will provide only 10 pixels of space. Use the following code to do that.

```
root.getColumnConstraints().add(new ColumnConstraints());
root.getColumnConstraints().add(new ColumnConstraints(10));
root.getColumnConstraints().add(new ColumnConstraints());
root.getColumnConstraints().add(new ColumnConstraints(10));
root.getColumnConstraints().add(new ColumnConstraints());
```

6. Set the padding to 10, set the `hgap` and `vgap` to 10, set the ID to `rootPane`, and make the default alignment centered.

```
        root.setId("rootPane");
        root.setVgap(10);
        root.setHgap(10);
        root.setPadding(new Insets(10, 10, 10, 10));
        root.setAlignment(Pos.CENTER);
```

7. Add a `children` element to the GridPane. Add additional elements within this `children` element.
8. Add a `Label` element with the following characteristics to the first row of the GridPane.
   a. Set the ID to `customerLabel`.
   b. Set the text to `Customer:`
   c. Place it in column 0, row 0.
   d. Here is the code to do that.

```

        Label customerLabel = new Label("Customer:");
        customerLabel.setId("customerLabel");

        root.add(customerLabel, 0, 0);
```

9. Add a `ChoiceBox` element with the following characteristics to the first row of the GridPane.
   a. Set the ID to `customerChoiceBox`.
   b. Place the `ChoiceBox` in column 2, row 0.
   c. The code is as follows:

```

customerChoiceBox.setId("customerChoiceBox");
root.add(customerChoiceBox, 2, 0);
```

10. Add a `Label` with following characteristics to the second row of the GridPane.
    a. Set the ID to `stockLabel`.
    b. Set the text to `Stock:`
    c. Place it in column 0, row 1.
    d. Use the following code:

```
        Label stockLabel = new Label("Stocks:");
        stockLabel.setId("stockLabel");

        root.add(stockLabel, 0, 1);
```

11. Add a `ChoiceBox` with the following characteristics to the second row of the GridPane.
    a. Set the ID to `stockChoiceBox`.
    b. Place it in column 2, row 1.
    c. Use the following code:

```
        stockChoiceBox.setId("stockChoiceBox");
        root.add(stockChoiceBox, 2, 1);
```

12. Add a `Label` with the following characteristics to the third row of the GridPane.
    a. Set the ID to `priceLabel`.
    b. Set the text to `Price:`
    c. Place it in column 0, row 2.
    d. Use the following code:

```
        Label priceLabel = new Label("Price:");
        priceLabel.setId("priceLabel");
        root.add(priceLabel, 0, 2);
```

13. Add a `TextField` with the following characteristics to the third row of the GridPane.
    a. Set ID to `priceTextLabel`.
    b. Place it in column 0, row 2.
    c. Use the following code:

```
        priceTextField.setId("priceTextField");
        root.add(priceTextField, 2, 2);
```

14. Add a `Label` element with the following characteristics to the fourth row of the GridPane.
    a. Set the ID to `qtyLabel`.
    b. Set the text to `Quantity:`
    c. Place it in column 0, row 3.
    d. Use the following code:

```
        Label qtyLabel = new Label("Quantity");
        qtyLabel.setId("qtyLabel");
        root.add(qtyLabel, 0, 3);
```

15. Add a `TextField` with the following characteristics to the fourth row of the GridPane.
    a. Set the ID to `quantityTextField`.
    b. Place it in column 2, row 3.
    c. Use the following code:

```
        quantityTextField.setId("quanitityTextfield");
        root.add(quantityTextField, 2, 3);
```

16. Add an `HBox` to the fifth row of the GridePane.
    a. Set the ID of the HBox to `buttonHBox` and put the element in column 2, row 4. Have it span two columns.
    b. Add a `children` element.
    c. Add two buttons inside the `children` element.
    d. Use the following code:

```
        HBox buttonBox = new HBox();
        Button saveBtn = new Button("Save");
        Button cancelBtn = new Button("Cancel");

        buttonBox.getChildren().add(saveBtn);
        buttonBox.getChildren().add(cancelBtn);
        buttonBox.setId("buttonHBox");

        root.add(buttonBox, 1, 4, 3, 1);
```

17. Configure the resultsTextArea text area and add it to row 6 of the GridPane. Have it span two columns.
    a. Set the text to "Sample Text".
    b. Set the preferred width to 300 and the preferred height to 100.
    c. Use the following code:

```
        resultsTextArea.setText("Sample text");
        resultsTextArea.setPrefWidth(300);
        resultsTextArea.setPrefHeight(100);

        root.add(resultsTextArea, 2, 5, 2, 1);
```

18. Save and Run the application. Correct any errors.
19. Open the `OrderDialog.css` file. Note the styles that are defined.
    a. Notice the CSS selectors and the rules and how they are defined in JavaFX.
    b. For the `rootPane`, note the background styling.
    c. Close the file.
20. Add the OrderDialog.css to the application. Use the following code in the location indicated in the comments.

```
scene.getStylesheets().add(OrderDialogApp.class.getResource("Ord
erDialog.css").toExternalForm());
```

21. Run the application and ensure that it looks like the image shown at the beginning of this practice.

Practice 5-2: (Summary Level) Adding Events to the Order Form

Overview

In this practice, you add event handlers to the order form.

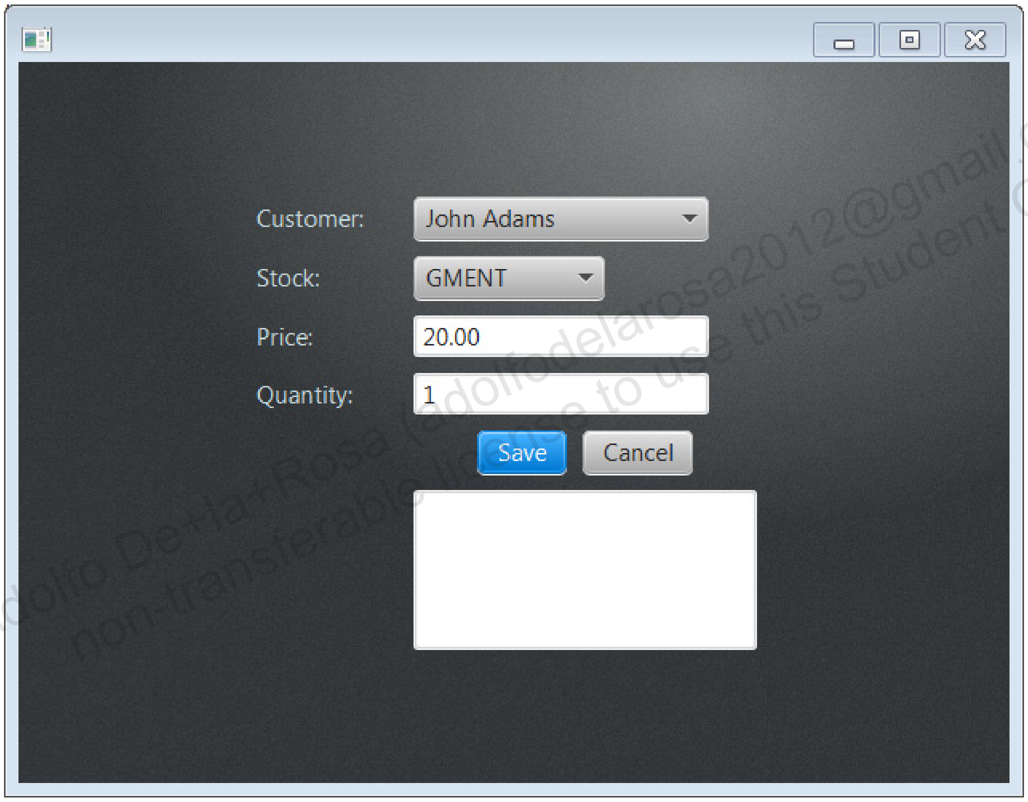
Assumptions

You have completed Practice 5-1.

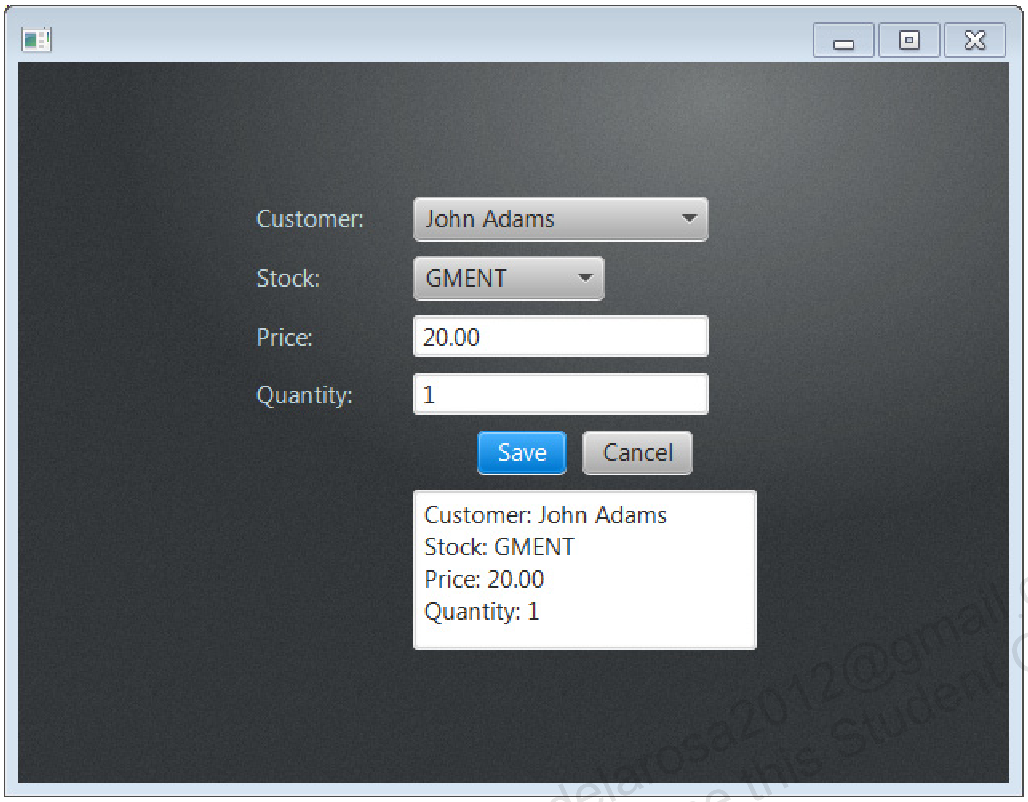
Summary

In this practice, you add events to the order form to test your user interface. Add a text area to the form to show which data you selected. When complete, your application should look like the following image.

The order form after the application loads:



The order form after clicking Save:



```
Customer: John Adams
Stock: GMENT
Price: 20.00
Quantity: 1
```

Tasks

1. Open the `OrderDialogEvt` project from the practice directory.
2. Make sure it is set as the Main Project in NetBeans.
3. Open the `OrderDialog.java` file.
4. The sample data to use in the form is provided for you.
5. Initialize the `stockChoiceBox` with the `stockNames` `ObservableList`.
   Hint: Use the code for `customerChoiceBox` as a model for your code.
6. Implement event handlers for the `SaveBtn` and `CancelBtn`.
7. Compile and run the application. Correct any errors.
8. Close the project.

Practice 5-2: (Detail Level) Adding Events to the Order Form

Overview

In this practice, you add event handlers to the order form.

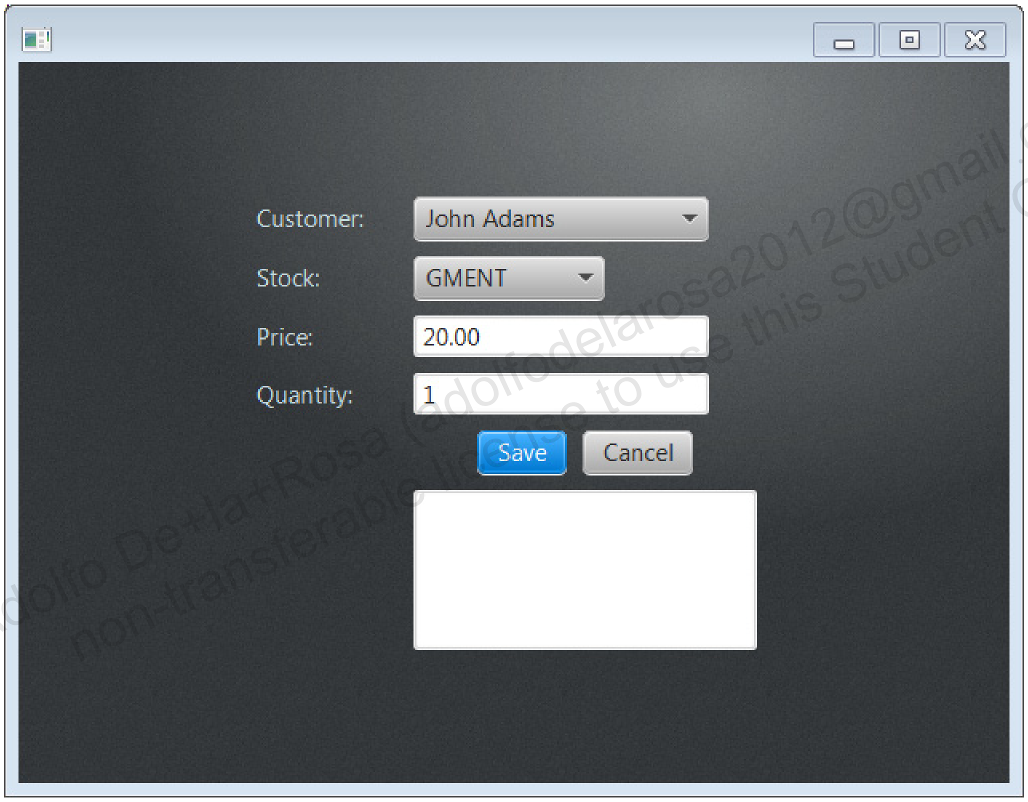
Assumptions

You have completed Practice 5-1.

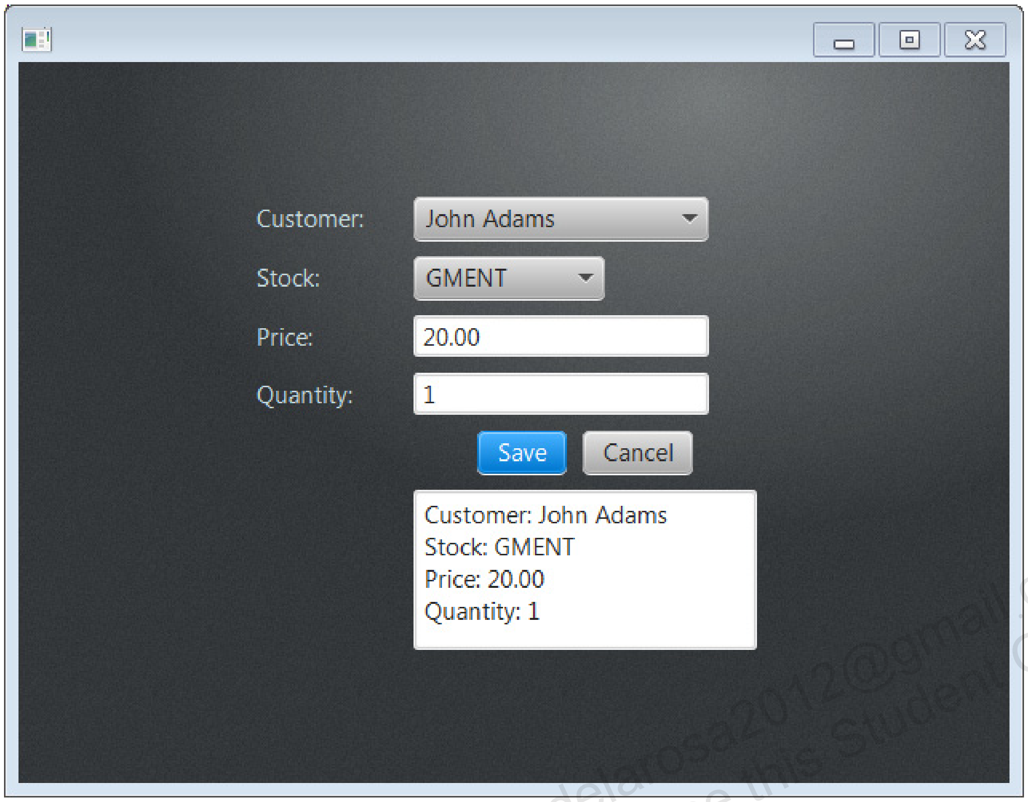
Summary

In this practice, you add events to the order form to test your user interface. Add a text area to the form to show which data you selected. When complete, your application should look like the following image.

The order form after the application loads:



The order form after clicking Save:



```
Customer: John Adams
Stock: GMENT
Price: 20.00
Quantity: 1
```

Tasks

1. Open the `OrderDialogEvt` project from the practice directory.
2. Make sure it is set as the Main Project in NetBeans.
3. Open the `OrderDialog.java` file.
4. The sample data to use in the form is provided for you.
5. Initialize the `stockChoiceBox` with the `stockNames` `ObservableList`.

   Hint: Use the code for `customerChoiceBox` as a model for your code.

   For example, use the following code.

```
stockChoiceBox.setItems(stockNames);
stockChoiceBox.getSelectionModel().selectFirst();
```

6. Implement event handlers for the `SaveBtn` and `CancelBtn`.
   Use the following code.

```
saveBtn.setOnAction(new EventHandler<ActionEvent>() {

  @Override
  public void handle(ActionEvent Event) {
    StringBuilder tempStr = new StringBuilder();
    tempStr.append("Customer:
").append(customers[customerChoiceBox.getSelectionModel().getSel
ectedIndex()]);
    tempStr.append("\nStock:
").append(stocks[stockChoiceBox.getSelectionModel().getSelectedI
ndex()]);
    tempStr.append("\nPrice:
").append(priceTextField.getText());

tempStr.append("\nQuantity:").append(quantityTextField.getText()
);


    resultsTextArea.setText(tempStr.toString());
  }
});

cancelBtn.setOnAction(new EventHandler<ActionEvent>() {

  @Override
  public void handle(ActionEvent Event) {
     customerChoiceBox.getSelectionModel().selectFirst();
     stockChoiceBox.getSelectionModel().selectFirst();
     priceTextField.setText("20.00");
     quantityTextField.setText("1");
     resultsTextArea.setText("Sample text");
   }
});
```

7. Compile and run the application. Correct any errors.
8. Close the project.

Practice 5-3: (Summary Level) Creating an Interactive Pie Chart

Overview

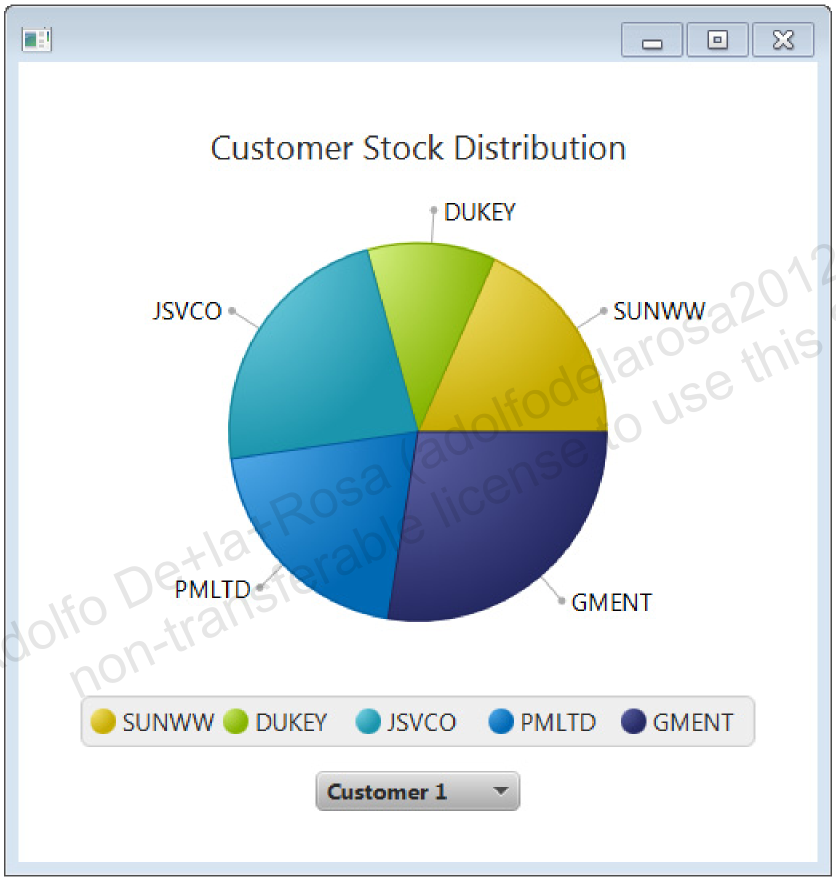
In this practice, you create a PieChart application that is automatically updated based on the selected customer.

Assumptions

You have listened to the lecture portion of this lesson and completed the previous two practices.

Summary

Complete the parts of the application indicated so that a `ChoiceBox` event handler will update the `PieChart` when a different customer is selected.


Customer Stock Distribution
DUKEY
JSVCO
SUNWW
PMLTD
GMENT
● SUNWW ● DUKEY ● JSVCO ● PMLTD ● GMENT
Customer 1

Tasks

1. Open the `PieChartApp` project for this lesson and practice from the student labs directory.
2. Make sure it is set as the Main Project in NetBeans.
3. Open the `PieChartApp.java` file.
4. Add an event handler to handle changes in the `ChoiceBox`.
5. Add code to the `setCode` method to update the `PieChart` when a different customer is selected.

   Hint: A `switch` statement might work well for this.
6. Build and run your program. Correct any errors.

Practice 5-3: (Detail Level) Creating an Interactive Pie Chart

Overview

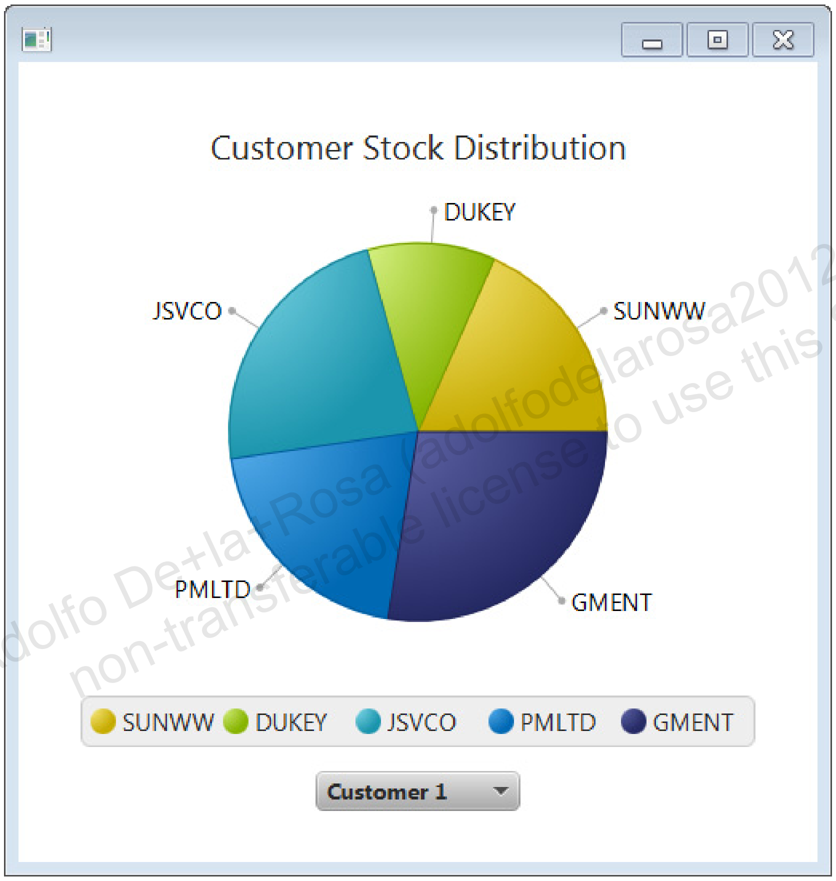
In this practice, you create a PieChart application that is automatically updated based on the selected customer.

Assumptions

You have listened to the lecture portion of this lesson and completed the previous two practices.

Summary

Complete the parts of the application indicated so that a `ChoiceBox` event handler will update the `PieChart` when a different customer is selected.



Tasks

1. Open the `PieChartApp` project for this lesson and practice from the student labs directory.
2. Make sure it is set as the Main Project in NetBeans.
3. Open the `PieChartApp.java` file.

4. Add an event handler to handle changes in the `ChoiceBox`.

   The suggested code might look like the following.

```
customerChoiceBox.getSelectionModel().selectedIndexProperty().ad
dListener(
            new ChangeListener<Number>() {

            public void changed(ObservableValue ov, Number
value, Number newValue) {
                    setChart(newValue.intValue());
            }
    });
```

5. Add code to the `setCode` method to update the `PieChart` when a different customer is selected.

   Hint: A `switch` statement might work well for this.

```
    private void setChart(int cbIndex) {
        String selectedItem = customerList.get(cbIndex);

        switch (selectedItem) {
            case "Customer 1":
                customerPieChart.setData(customer01Data);
                break;
            case "Customer 2":
                customerPieChart.setData(customer02Data);
                break;
            default:
                customerPieChart.setData(customer01Data);

        }
    }
```

6. Build and run your program. Correct any errors.

Practices for Lesson 6: Visual Effects, Animation, WebView, and Media

Chapter 6

Practice for Lesson 6

Practice Overview

In the practice for this lesson, you add transitions to your application.

Practice 6-1: (Summary Level) Adding a Fade Transition

Overview

In this practice, you create an application that fades a picture in and out of your application.

Assumptions

You have completed the lecture portion of the lesson.

Summary

Add the code necessary to provide the desired effects.

Tasks

1. Open the `TransitionApp` project in the `practice01` directory for this project.
2. Build and run the application. Note that the picture is displayed, but there are no transitions.
3. Add a transition that completely hides the picture when the Hide button is clicked. See the Ensemble application for examples of FadeTransitionBuilder. The transition should have the following characteristics:
   a. From value is `1`.
   b. To value is `0`.
   c. Cycle count is `1`.
   d. AutoReverse should be set to `false`.
4. Add a transition that completely shows the picture when the Show button is clicked. The transition should have the following characteristics:
   a. From value is `0`.
   b. To value is `1`.
   c. Cycle count is `1`.
   d. AutoReverse should be set to `false`.
5. Build and run your application. Correct any errors you encounter.

Practice 6-1: (Detail Level) Adding a Fade Transition

Overview

In this practice, you create an application that fades a picture in and out of your application.

Assumptions

You have completed the lecture portion of the lesson.

Summary

Add the code necessary to provide the desired effects.

Tasks

1. Open the `TransitionApp` project in the `practice01` directory for this project.
2. Build and run the application. Note that the picture is displayed, but there are no transitions.
3. Add a transition that completely hides the picture when the Hide button is clicked.

```
        hideTransition = FadeTransitionBuilder.create()
            .duration(Duration.seconds(4))
            .node(pic01)
            .fromValue(1)
            .toValue(0.0)
            .cycleCount(1)
            .autoReverse(false)
            .build();
```

4. Add a transition that completely shows the picture when the Show button is clicked.

```
        showTransition = FadeTransitionBuilder.create()
            .duration(Duration.seconds(4))
            .node(pic01)
            .fromValue(0)
            .toValue(1)
            .cycleCount(1)
            .autoReverse(false)
            .build();
```

5. Build and run your application. Correct any errors you encounter.

Practices for Lesson 7: JavaFX Tables and Client GUI

Chapter 7

Practice for Lesson 7

Practices Overview

In these practices, you create and style a JavaFX TableView.

Practice 7-1: (Summary Level) Creating a Simple TableView

Overview

In this practice, you use JavaFX to create a TableView.

Assumptions

You have completed the lecture portion of this lesson.

Summary

Create a JavaFX TableView that changes based on changes in a ChoiceBox. Add the necessary controls and event handlers.

Tasks

1. Open the `DataTable` project located in the `practice01` directory for this lesson.
2. Edit the `DataTable.java` file.
3. Review the UI controls defined at the start of the class. You will be using initializing these fields in the methods that follow.
4. Make the following edits in the `init` method.
5. For the `mainBox` `VBox`, set the spacing to 10. Set the alignment to center. Set the padding to 10.
6. Set the data value of the `customerChoiceBox` to `customerList01`. Highlight the first item in the `ChoiceBox`.
7. Create an event handler that will update the table data based on the customer selected.
8. Create a label for the "Stocks" table. Use a 16 pt font with a "bold" weight.
9. Set up the table columns.
10. Associate the Share object fields with the columns.
11. Add columns and initialize table.
12. Add `Label`, `TableView`, and `ChoiceBox` to the `VBox`.
13. Add a `setTable` method to change the data in the `TableView`.
14. Build and run the application. Correct any errors you may encounter.

Practice 7-1: (Detail Level) Creating a Simple TableView

Overview

In this practice, you use JavaFX to create a TableView.

Assumptions

You have completed the lecture portion of this lesson.

Summary

Create a JavaFX TableView that changes based on changes in a ChoiceBox. Add the necessary controls and event handlers.

Tasks

1. Open the `DataTable` project located in the `practice01` directory for this lesson.
2. Edit the `DataTable.java` file.
3. Review the UI controls defined at the start of the class. You will be using initializing these fields in the methods that follow.
4. Make the following edits in the `init` method.
5. For the `mainBox` `VBox`, set the spacing to 10. Set the alignment to center. Set the padding to 10.

```
mainBox.setSpacing(10);
mainBox.setAlignment(Pos.CENTER);
mainBox.setPadding(new Insets(10, 10, 10, 10));
```

6. Set the data value of the `customerChoiceBox` to `customerList01`. Highlight the first item in the `ChoiceBox`.

```
customerChoiceBox.setItems(customerList01);
customerChoiceBox.getSelectionModel().selectFirst();
```

7. Create an event handler that will update the table data based on the customer selected.

```

customerChoiceBox.getSelectionModel().selectedIndexProperty().ad
dListener(
    new ChangeListener<Number>(){
        @Override
        public void changed(ObservableValue ov, Number value,
Number newValue ){
            setTable(newValue.intValue());
    }
});
```

8. Create a label for the "Stocks" table. Use a 16 pt font with a "bold" weight.

```
Label centerLbl = new Label("Stocks");
centerLbl.setStyle("-fx-font-size:16pt; -fx-font-weight:bold;");
```

9. Set up the table columns.

```
TableColumn symbolCol = new TableColumn("Symbol");
TableColumn quantCol = new TableColumn("Quantity");
TableColumn purchDateCol = new TableColumn("Date");
TableColumn purchPriceCol = new TableColumn("Price");
TableColumn sectorCol = new TableColumn("Sector");
```

10. Associate the Share object fields with the columns.

```
symbolCol.setCellValueFactory(new PropertyValueFactory<Shares,
String>("symbol"));
quantCol.setCellValueFactory(new PropertyValueFactory<Shares,
String>("quantity"));
purchDateCol.setCellValueFactory(new
PropertyValueFactory<Shares, String>("purchaseDate"));
purchPriceCol.setCellValueFactory(new
PropertyValueFactory<Shares, String>("purchasePrice"));
sectorCol.setCellValueFactory(new PropertyValueFactory<Shares,
String>("sector"));
```

11. Add columns and initialize table.

```
table.getColumns().addAll(symbolCol, quantCol, purchDateCol,
purchPriceCol, sectorCol);
table.setItems(customer1);
```

12. Add `Label`, `TableView`, and `ChoiceBox` to the `VBox`.

```
mainBox.getChildren().add(centerLbl);
mainBox.getChildren().add(table);
mainBox.getChildren().add(customerChoiceBox);
```

13. Add a `setTable` method to change the data in the `TableView`.

```
    public void setTable(int selection){
        switch (selection){
            case 0:
                table.setItems(customer1);
                break;
            case 1:
                table.setItems(customer2);
                break;
            case 2:
                table.setItems(customer3);
                break;
            default:
                table.setItems(customer1);
        }
    }
```

14. Build and run the application. Correct any errors you may encounter.

Practice 7-2: (Summary Level) Styling a Smart Table

Overview

In this practice, you use CSS to style the TableView you created in the previous practice.

Assumptions

You completed the lecture for this lesson and the previous practice.

Summary

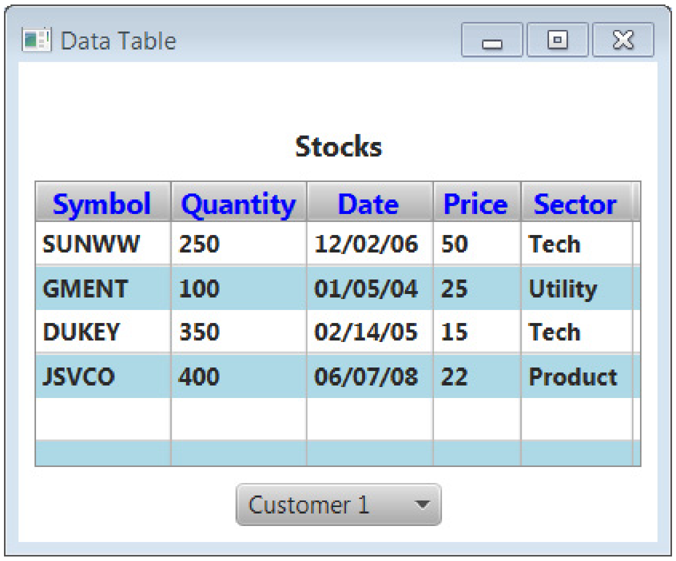
Add a CSS file to your application. Style the table using CSS.

Tasks

1. Open the `DataTableStyled` project located in the `practice02` directory for this lesson or continue working with your application from the previous exercise.
2. Edit the `DataTable.java` file.
3. Add a statement to style your application with `DataTable.css`.
4. Create or edit the `DataTable.css` file and add CSS styles to the selectors in the file. Examples of CSS values you can use for each rule are as follows:

```
-fx-text-fill:blue;
-fx-font-weight:bold;
-fx-alignment:center;
-fx-background-color:lightblue;
-fx-font-size:18pt;
-fx-spacing:10px;
-fx-alignment:center;
-fx-padding:10px;
```

5. Try to make your styled table look like the following:



Stocks

Symbol	Quantity	Date	Price	Sector
SUNWW	250	12/02/06	50	Tech
GMENT	100	01/05/04	25	Utility
DUKEY	350	02/14/05	15	Tech
JSVCO	400	06/07/08	22	Product

6. Build and run your application. Correct any errors you encounter.

Practice 7-2: (Detail Level) Styling a Smart Table

Overview

In this practice, you use CSS to style the TableView you created in the previous practice.

Assumptions

You completed the lecture for this lesson and the previous practice.

Summary

Add a CSS file to your application. Style the table using CSS.

Tasks

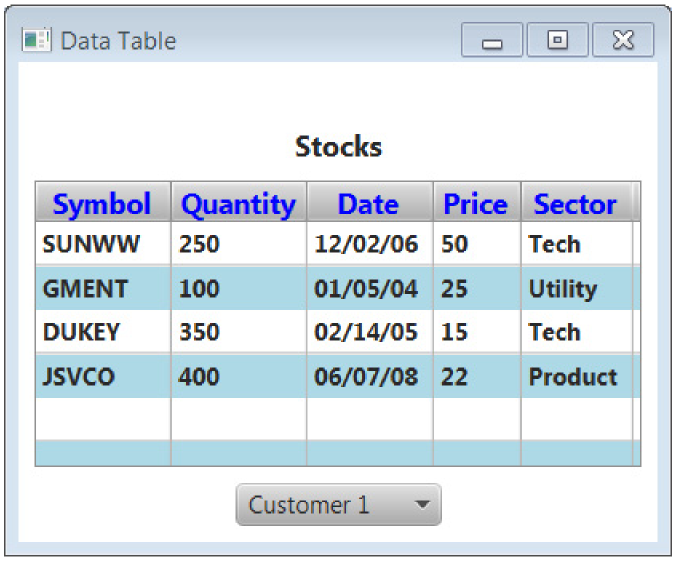
1. Open the `DataTableStyled` project located in the `practice02` directory for this lesson or continue working with your application from the previous exercise.
2. Edit the `DataTable.java` file.
3. Add a statement to style your application with `DataTable.css`.

```
scene.getStylesheets().add(DataTable.class.getResource("DataTabl
e.css").toExternalForm());
```

4. Create or edit the `DataTable.css` file and add CSS styles to the selectors in the file. Add the following CSS rules.

```
#stockTable .label{
    -fx-text-fill:blue;
    -fx-font-weight:bold;
}

#stockTable .table-row-cell{
    -fx-font-weight:bold;
    -fx-alignment:center;
}

#stockTable .table-row-cell:odd{
    -fx-background-color:lightblue;
}

#centerLbl{
    -fx-font-weight:bold;
    -fx-font-size:18pt;
}

#mainBox{
    -fx-spacing:10px;
    -fx-alignment:center;
    -fx-padding:10px;
}
```

5. Try to make your styled table look like the following:



Stocks

Symbol	Quantity	Date	Price	Sector
SUNWW	250	12/02/06	50	Tech
GMENT	100	01/05/04	25	Utility
DUKEY	350	02/14/05	15	Tech
JSVCO	400	06/07/08	22	Product

6. Build and run your application. Correct any errors you encounter.

Practice 7-3: (Summary Level) Creating a Complete BrokerTool Interface

Overview

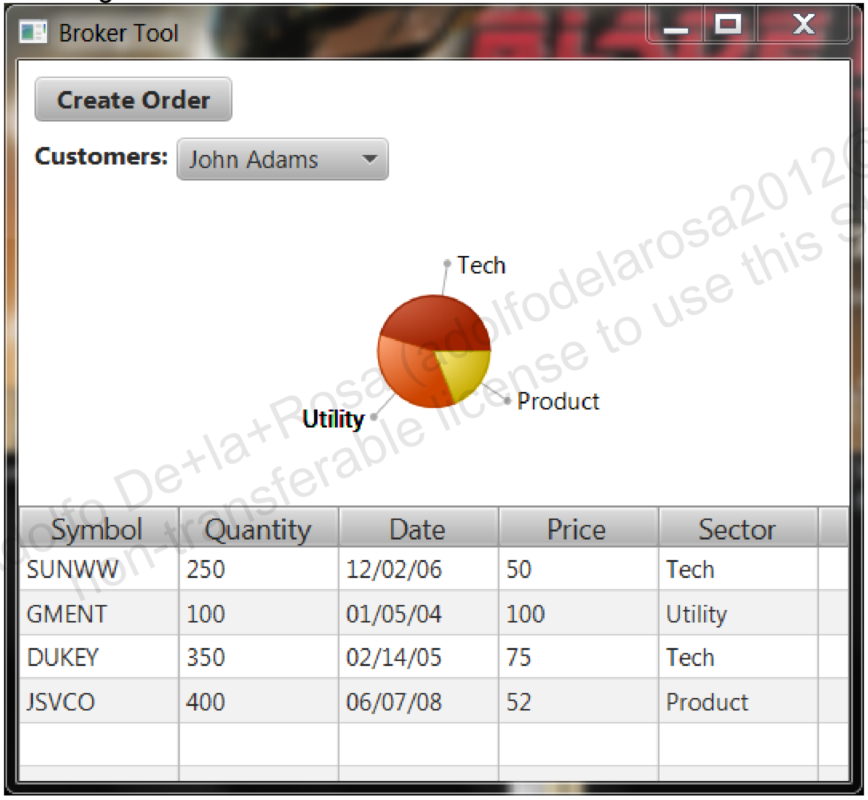
In this practice, you create a complete mock BrokerTool graphical user interface.

Assumptions

You have completed the lecture portion of this lesson. In addition, you have completed the preceding practices.

Summary

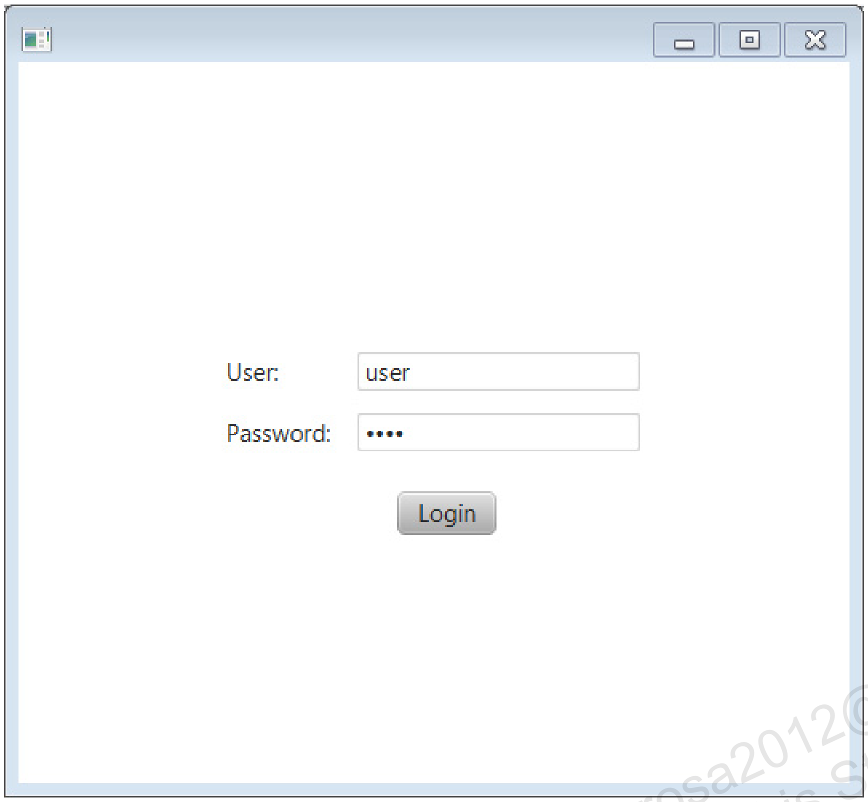
You will take a skeleton application, which includes some nice transitions, and you will build out a full mock BrokerTool interface. When complete, your application should look and flow like the following:



Symbol	Quantity	Date	Price	Sector
SUNWW	250	12/02/06	50	Tech
GMENT	100	01/05/04	100	Utility
DUKEY	350	02/14/05	75	Tech
JSVCO	400	06/07/08	52	Product

Login Screen

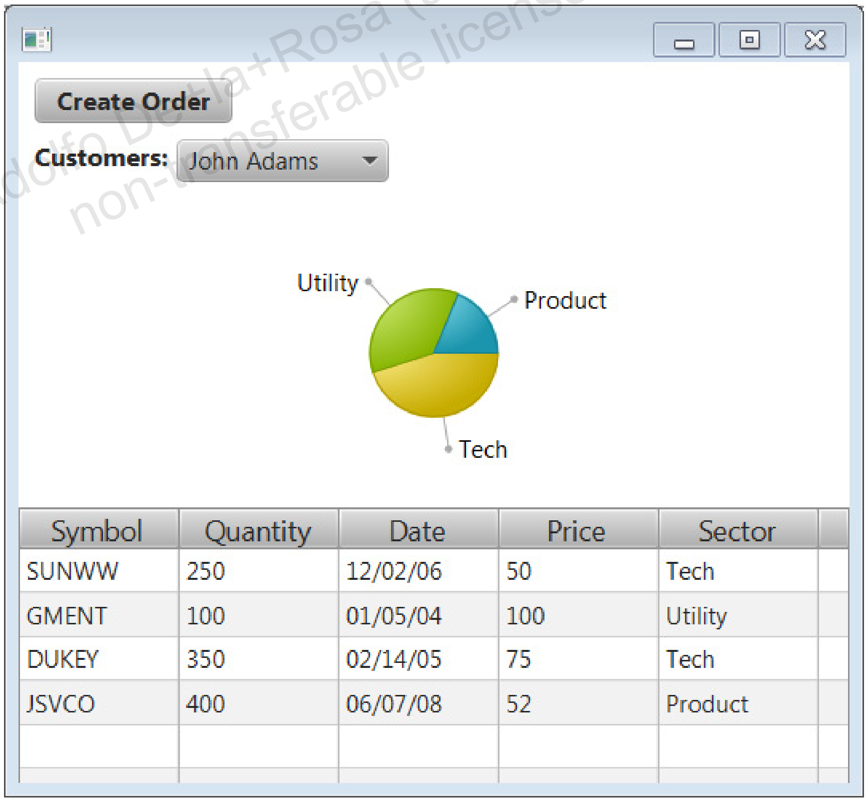
When you start the application, you should land on this pane. Clicking the **Login** button takes you to the next screen.


User: user

Password: ••••

DashBoard

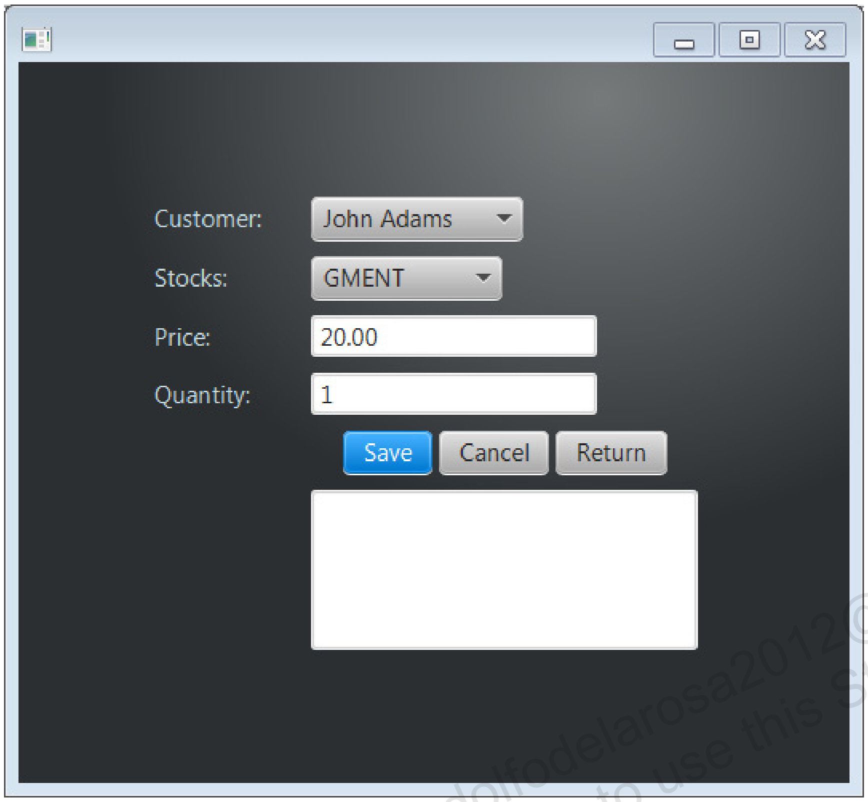
The DashBoard should look like this. Click the **Order Form** button to go to the next screen.



Symbol	Quantity	Date	Price	Sector
SUNWW	250	12/02/06	50	Tech
GMENT	100	01/05/04	100	Utility
DUKEY	350	02/14/05	75	Tech
JSVCO	400	06/07/08	52	Product

Order Form

Use this page for orders. Click the **Return** button to return to the DashBoard.



Tasks

1. Open the `BrokerToolGui` project located in the `practice03` directory for this lesson.
2. Build and run the application.
3. Review the UI elements and determine what is missing from each screen.
4. Using the previous projects you have created, update the three screens so they look like those presented in the Summary.
5. Correct any errors you may encounter.

Practice 7-3: (Detail Level) Creating a Complete BrokerTool Interface

Overview

In this practice, you create a complete mock BrokerTool graphical user interface.

Assumptions

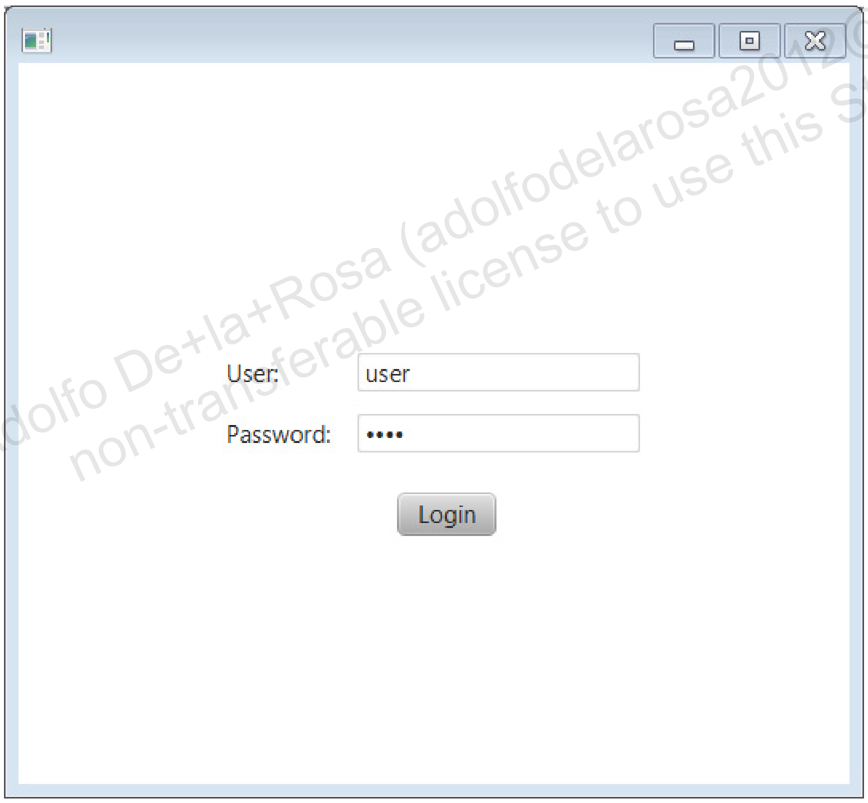
You have completed the lecture portion of this lesson. In addition, you have completed the preceding practices.

Summary

You will take a skeleton application, which includes some nice transitions, and you will build out a full mock BrokerTool interface. When complete, your application should look and flow like the following:

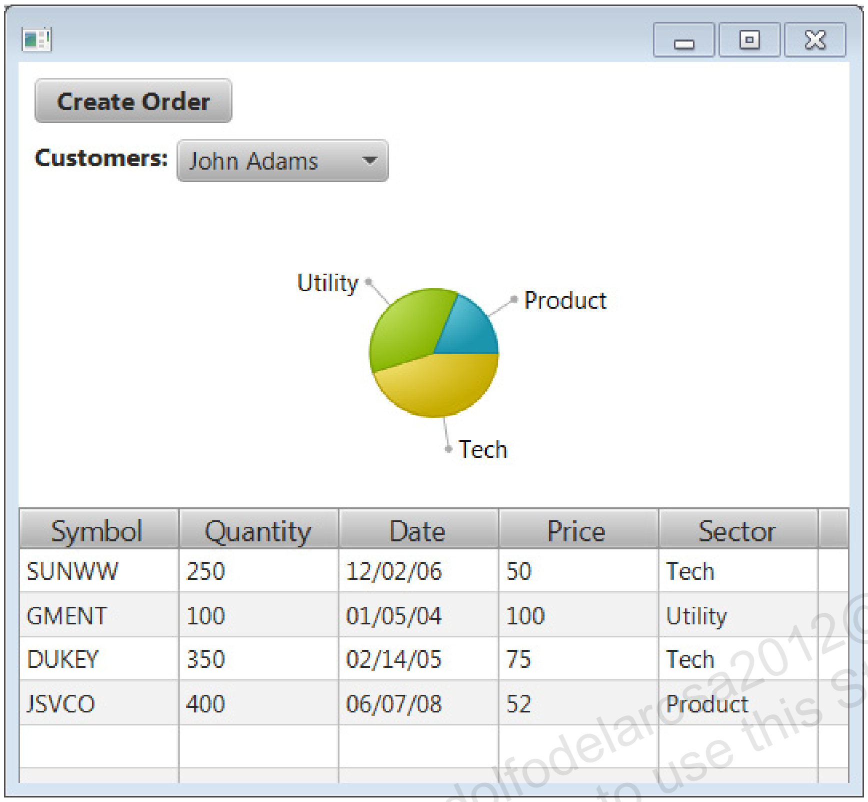
Login Screen

When you start the application you should land on this pane. Clicking the **Login** button takes you to the next screen.



DashBoard

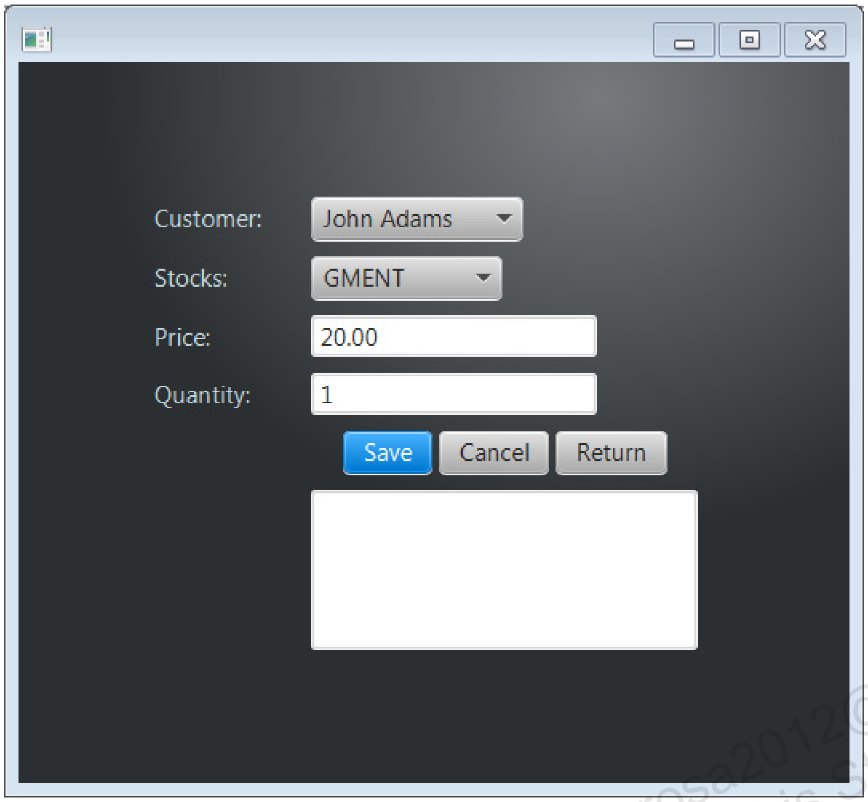
The DashBoard should look like this. Click the **Order Form** button to go to the next screen.



Symbol	Quantity	Date	Price	Sector
SUNWW	250	12/02/06	50	Tech
GMENT	100	01/05/04	100	Utility
DUKEY	350	02/14/05	75	Tech
JSVCO	400	06/07/08	52	Product

Order Form

Use this page for orders. Click the **Return** button to return to the DashBoard.



Tasks

1. Open the `BrokerToolGui` project located in the `practice03` directory for this lesson.
2. Build and run the application.
3. Review the UI elements and determine what is missing from each screen.
4. Using the previous projects you have created, update the three screens so they look like those presented in the Summary.
5. Add the code to `buildLogin` method to complete the login screen.

```
        TextField userTextField = new TextField();
        loginPane.add(userTextField, 2, 2);
        Label pwdLabel = new Label("Password:");
        loginPane.add(pwdLabel, 1, 3);
        PasswordField pwdField = new PasswordField();
        loginPane.add(pwdField, 2, 3);
```

6. Add the following code to the `buildDashBoard` method to build the stock dashboard.

```
        // Set up choice box
        Label customersLabel = new Label("Customers: ");
        customersLabel.setId("dashCustomersLbl");
        customerChoiceBox.setItems(customerList);
        customerChoiceBox.getSelectionModel().selectFirst();
        customerChoiceBox.setId("customerChoiceBox");
        // ChoiceBox Event handler
        customerChoiceBox.getSelectionModel()
.selectedIndexProperty().addListener(
            new ChangeListener<Number>() {

            @Override
            public void changed(ObservableValue ov, Number
value, Number newValue) {
                System.out.println("Customer Selected");
                setChart(newValue.intValue());
            }
        });


        // Set up Chart
        customerPieChart.setData(customer01Data);
        customerPieChart.setClockwise(false);
        customerPieChart.setLegendVisible(false);

        // Setup Table
        Label centerLbl = new Label("Stocks");
        centerLbl.setId("centerLbl");

        TableColumn symbolCol = new TableColumn("Symbol");
        TableColumn quantCol = new TableColumn("Quantity");
        TableColumn purchDateCol = new TableColumn("Date");
        TableColumn purchPriceCol = new TableColumn("Price");
        TableColumn sectorCol = new TableColumn("Sector");

        symbolCol.setPrefWidth(100);
        quantCol.setPrefWidth(100);
        purchDateCol.setPrefWidth(100);
        purchPriceCol.setPrefWidth(100);
        sectorCol.setPrefWidth(100);

        // Associate CustShares object with table columnds
        symbolCol.setCellValueFactory(
                new PropertyValueFactory<Shares,
String>("symbol"));
        quantCol.setCellValueFactory(
                new PropertyValueFactory<Shares,
String>("quantity"));
        purchDateCol.setCellValueFactory(
                new PropertyValueFactory<Shares,
String>("purchaseDate"));
        purchPriceCol.setCellValueFactory(
                new PropertyValueFactory<Shares,
String>("purchasePrice"));
        sectorCol.setCellValueFactory(
                new PropertyValueFactory<Shares,
String>("sector"));

        table.getColumns().addAll(symbolCol, quantCol,
purchDateCol, purchPriceCol, sectorCol);

        table.setItems(customer1); // Init table
        table.setId("stockTable");


        // Add page elements to the stage
        customersBox.getChildren().add(customersLabel);
        customersBox.getChildren().add(customerChoiceBox);
        dashButtonBox.getChildren().add(customersBox);

        centerBox.getChildren().add(customerPieChart);
        centerBox.getChildren().add(table);
```

7. Add the code to build rows 2 thru 4 of the grid pane.

```
        // Row 2
        Label stockLabel = new Label("Stocks:");
        stockLabel.setId("stockLabel");
        stockChoiceBox.setItems(stockList);
        stockChoiceBox.getSelectionModel().selectFirst();

        orderPane.add(stockLabel, 0, 1);
        orderPane.add(stockChoiceBox, 2, 1);

        // Row 3
        Label priceLabel = new Label("Price:");
        priceLabel.setId("priceLabel");
        orderPane.add(priceLabel, 0, 2);
        orderPane.add(priceTextField, 2, 2);

        // Row 4
        Label qtyLabel = new Label("Quantity");
        qtyLabel.setId("qtyLabel");
        orderPane.add(qtyLabel, 0, 3);
        orderPane.add(quantityTextField, 2, 3);
```

8. Add code for the `Save` and `Cancel` Buttons.

```
        Button saveBtn = new Button("Save");
        Button cancelBtn = new Button("Cancel");

        orderButtonBox.getChildren().add(saveBtn);
        orderButtonBox.getChildren().add(cancelBtn);
```

9. Add code for grid pane row 6.

```
// Row 6

resultsTextArea.setText("Sample text");
resultsTextArea.setPrefWidth(300);
resultsTextArea.setPrefHeight(100);

orderPane.add(resultsTextArea, 2, 5, 2, 1);
```

10. Add event handlers for the Save and Cancel buttons.

```
// Add event handlers
saveBtn.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent Event) {
       StringBuilder tempStr = new StringBuilder();
       tempStr.append("Customer:
").append(customerList.toArray()[customerOrderChoiceBox.getSelec
tionModel().getSelectedIndex()]);
       tempStr.append("\nStock:
").append(stockList.toArray()[stockChoiceBox.getSelectionModel()
.getSelectedIndex()]);
       tempStr.append("\nPrice:
").append(priceTextField.getText());
       tempStr.append("\nQuantity:
").append(quantityTextField.getText());

       resultsTextArea.setText(tempStr.toString());
   }
});

cancelBtn.setOnAction(new EventHandler<ActionEvent>() {

   @Override
   public void handle(ActionEvent Event) {
      customerOrderChoiceBox.getSelectionModel().selectFirst();
      stockChoiceBox.getSelectionModel().selectFirst();
      priceTextField.setText("20.00");
      quantityTextField.setText("1");
      resultsTextArea.setText("Sample text");
   }
});
```

11. Add the code to link the CSS style sheet for the application.

```
scene.getStylesheets().add(BrokerToolApp.class.getResource("Brok
erToolApp.css").toExternalForm());
```

12. Add the following CSS rules to style your application.

```
#orderPane{
    -fx-padding:0;
    -fx-background-color: radial-gradient(center 70% 5%, radius
60%, #767a7b,#2b2f32);
    -fx-background-image: url("noise.png");
    -fx-background-repeat: repeat;
    -fx-background-position: left top;
}

#customerLabel,#stockLabel,#priceLabel,#qtyLabel{
   -fx-text-fill: #c4d8de;
}

#orderButtonHBox{
    -fx-spacing:4px;
    -fx-alignment:center;
}

/*  dashBoardPane */

#dashButtonBox{
    -fx-spacing:10px;
    -fx-padding:10px;

}

#centerBox{
    -fx-spacing:10px;
}
```

```
#dashCustomersLbl, #orderButton{
    -fx-font-weight:bold;
    -fx-font-size:16pt;
}
```

13. Build and run your application.
14. Correct any errors you may encounter.

Practices for Lesson 8: JavaFX Concurrency and Binding

Chapter 8

Practice for Lesson 8

Practices Overview

In the practices for this lesson, you update the concurrency examples shown in the lecture to display the state of the `CounterService` in the user interface. After that, you add a second `Service` instance to your application.

Practice 8-1: (Summary Level) Displaying Service State Information

Overview

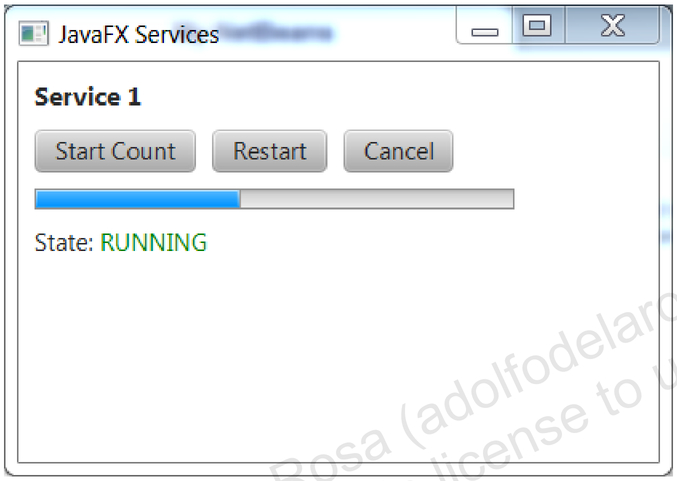
Based on the example application shown in the lecture, you update the application to display the current state of the service while it is executing.

Assumptions

You have completed the lecture portion of this lesson.

Summary

You update the provided project to monitor the state of your service. When complete, your application should look like the following when it executes:



Tasks

1. Open the `CountStateService` project in the `practice01` directory for this lesson.
2. Edit the `CounterService` class so that it returns a `CounterTask`.
3. Update the `CountBarApp` so that information about the services state is updated in the user interface. See the event handler for the start button.
4. Update the cancel button and restart button event handlers to appropriately change the state of the service. See the comments in the code for where to put your code.
5. Build and run your application. Correct any errors you may encounter.

Practice 8-1: (Detail Level) Displaying Service State Information

Overview

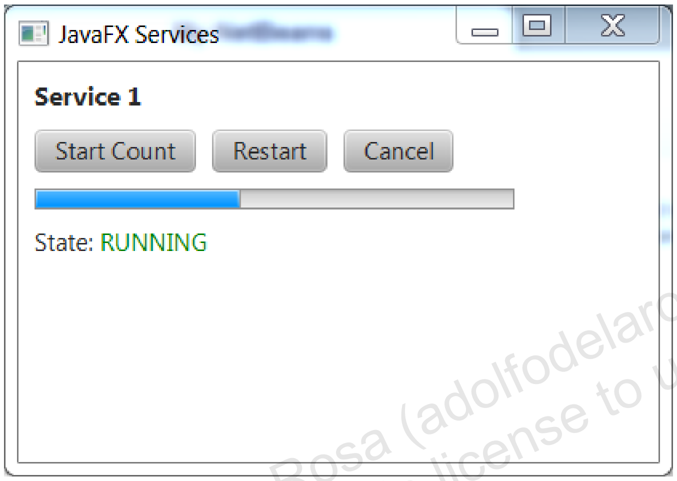
Based on the example application shown in the lecture, you update the application to display the current state of the service while it is executing.

Assumptions

You have completed the lecture portion of this lesson.

Summary

You update the provided project to monitor the state of your service. When complete, your application should look like the following when it executes:



State: RUNNING

Tasks

1. Open the `CountStateService` project in the `practice01` directory for this lesson.
2. Edit the `CounterService` class so that it returns a `CounterTask`.

```
    @Override
    protected Task<Void> createTask(){
        CounterTask ct = new CounterTask();
        return ct;
    }
```

3. Update the `CountBarApp` so that information about the services state is updated in the user interface. See the event handler for the start button.

```
startBtn01.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
        System.out.println("Count Started");
        bar.progressProperty().bind(cs.progressProperty());
        if (cs.getState() == State.READY) {
            cs.start();
            // Add Listener to the Service
            cs.stateProperty().addListener(new
ChangeListener<State>() {

                public void changed(ObservableValue<? extends
State> source, State oldState, State newState) {
                    switch (newState) {
                        case READY:
                            currentState.setText("READY");
                            currentState.setStyle("-fx-text-
fill:Green;");
                            break;

                        case SCHEDULED:
                            currentState.setText("SCHEDULED");
                            currentState.setStyle("-fx-text-
fill:Green;");
                            break;

                        case RUNNING:
                            currentState.setText("RUNNING");
                            currentState.setStyle("-fx-text-
fill:Green;");
                            break;

                        case SUCCEEDED:
                            currentState.setText("SUCCEEDED");
                            currentState.setStyle("-fx-text-
fill:Blue;");
                            break;

                        case FAILED:
                            currentState.setText("FAILED");
                            currentState.setStyle("-fx-text-
fill:Red;");
                            break;

                        case CANCELLED:
                            currentState.setText("CANCELLED");
                            currentState.setStyle("-fx-text-
fill:Red;");
```

```
                            break;

                        default:
                            currentState.setText("Not Set");
                            currentState.setStyle("-fx-text-
fill:Black;");
                            break;

                            }
                        }
                    });

                }
            }
        });
```

4. Update the cancel button and restart button event handlers to appropriately change the state of the service. See the comments in the code for where to put your code.

```
        cancelBtn01.setOnAction(new EventHandler<ActionEvent>()
{

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Count Stopped");
                cs.cancel();
                bar.progressProperty().unbind(); // Uncomment me
            }
        });

        restartBtn01.setOnAction(new EventHandler<ActionEvent>()
{
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Count Started");

bar.progressProperty().bind(cs.progressProperty());
                cs.restart();
            }
        });
```

5. Build and run your application. Correct any errors you may encounter.

Practice 8-2: (Summary Level) Adding a Second Service to your Application

Overview

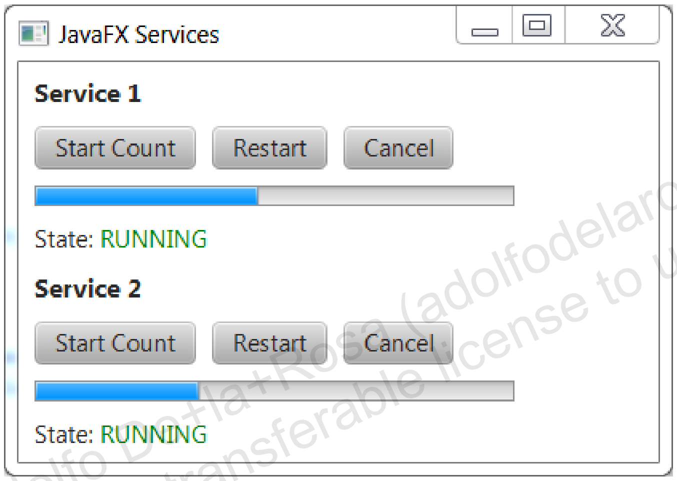
Now that you have an application performing concurrent tasks, you add a second service to application. Execute two instances of the service to make sure your application is executing concurrently.

Assumptions

You have completed the lecture portion of this lesson and the previous lesson.

Summary

You update your application to include a second CounterService. Then you run your application and make sure both services can execute at the same time. When done, your application should look like the following:



Tasks

1. Continue working with the project from the previous practice, or open the `CounterStateService` project in the `practice02` directory.
2. Add a second `CounterService`, associated UI controls, and event handlers to the application.
3. Build and run your application. Correct any errors you may encounter.

Practice 8-2: (Detail Level) Adding a Second Service to your Application

Overview

There are no detailed instructions for this practice.

Practices for Lesson 9: Java Persistence API (JPA)

Chapter 9

Practices for Lesson 9

Practices Overview

In these practices, you do the following:

- Create Entity classes by using JPA.
- Implement CRUD operations by using JPA.

Practice 9-1: (Summary Level) Creating Entity Classes by Using JPA

Overview

In this practice, you create and configure a persistent unit by doing the following:

- Create a database.
- Connect to the database.
- Generate Entity classes for tables in the database

Assumptions

NetBeans is running.

Tasks

1. Create the `playerDB` database in Java DB
   a. Enter the following information for the database:
      1) Database Name: `playerDB`
      2) User Name: `john`
      3) Password: `john`
      4) Click OK.
   b. Connect to the `playerDB` database.
   c. Execute the `playerDB.sql` file provided in the `D:\labs\09-IntroJPA\resources\` directory.
2. Examine the contents of the database. `Player` and `Team` tables will be created under the `John` schema.
3. Expand the Tables node to see the PLAYER and TEAM tables.
4. Create a new Java project, and name the project `Persistence`.
   a. Create Entity classes by completing the following steps:
      1) Right-click the `Persistence` project and select New > "Entity Classes from Database."
      2) Enter the following information to create Entity classes:
         - In the Database Connection field, select `jdbc:derby://localhost:1527/playerDB` from the drop-down list.
         - You see PLAYER and TEAM tables in the Available Table category.
         - Click Add All.
         - You see the PLAYER and TEAM tables in the Selected Table Category.
         - Click Next.
         - In the Entity Classes window, enter the Package Name as `data` and click Next.
         - In the Mappings Window, click Finish with the default selection.

5. Verify the creation of Entity Classes.
   a. Select the `Persistence` project.
   b. Open the data package, where you should see `Team.java` and `Player.java` created.
6. Open `persistence.xml` and name the persistence unit `PersistencePU`. An application's persistence unit is defined in a configuration file called `persistence.xml`. This file exists in the `META-INF` directory.

   To resolve any reference problems, add the `derbyclient.jar` file to the project library located in the `D:\labs\09-IntroJPA\resources folder`. You may find the file in your JDK installation location as well.

Practice 9-1: (Detail Level) Creating Entity Classes by Using JPA

Overview

In this practice, you create and configure a persistent unit by doing the following:

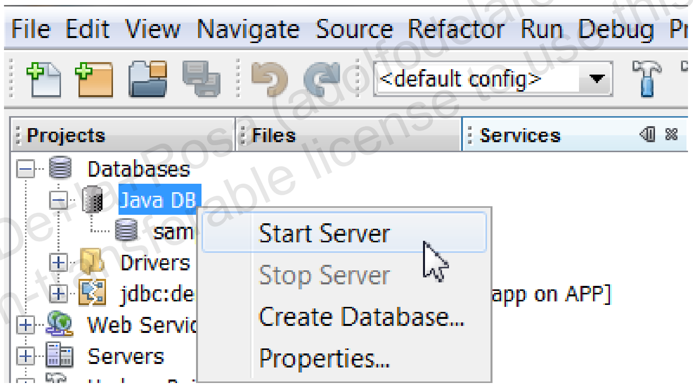
- Create a database.
- Connect to the database.
- Generate Entity classes for tables in the database.

Assumptions

NetBeans is running.

Tasks

1. Create the `playerDB` database.
   a. Start the Java DB database server from NetBeans.
      1) Click the Services tab.
      2) Expand the Databases node.
      3) Right-click the Java DB icon.
      4) Select Start Server.

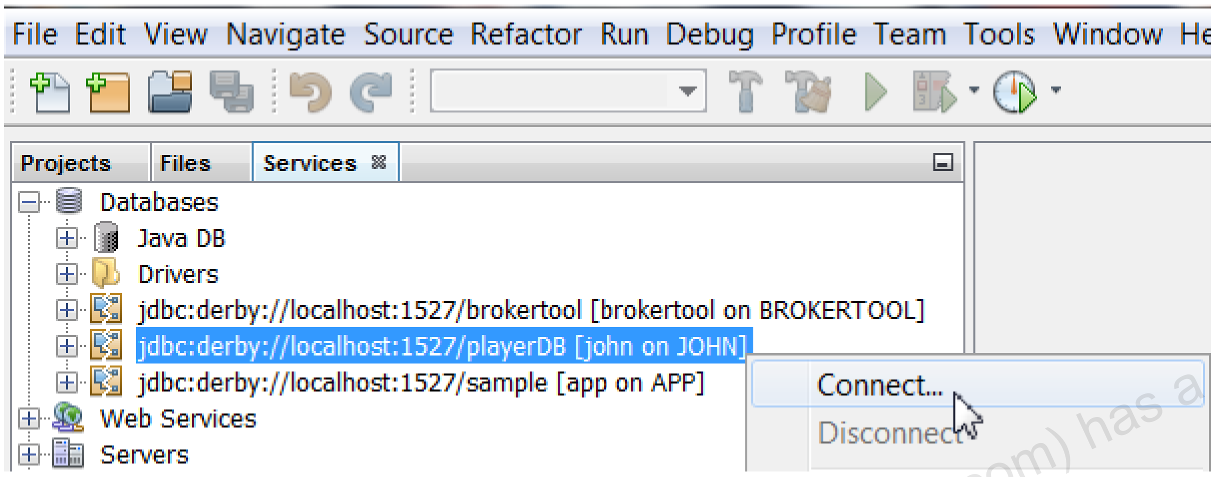


   b. Right-click the Java DB icon and select Create Database.
   c. Enter the following information for the database:
      1) Database Name: `playerDB`
      2) User Name: `john`
      3) Password: `john`
      4) Click OK.

      This creates the database and adds a connection for the database under the Databases icon.

d. Connect to the newly created database:

Right-click the `jdbc:derby://localhost:1527/playerDB` connection and select Connect.


File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window He
Projects
Files
Services
Databases
Java DB
Drivers
jdbc:derby://localhost:1527/brokertool [brokertool on BROKERTOOL]
jdbc:derby://localhost:1527/playerDB [john on JOHN]
jdbc:derby://localhost:1527/sample [app on APP]
Web Services
Servers
Connect...
Disconnect

e. From within NetBeans, select File > Open File and choose the `playerDB.sql` file provided in the `D:\labs\09-IntroJPA\resources\playerDB.sql` directory.

f. On the `playerDB.sql` tab, select `jdbc:derby://localhost:1527/playerDB` as the connection.

g. Click the Run SQL icon to execute the SQL statement.

2. Examine the contents of the database.

a. In the Services window, expand the `jdbc:derby://localhost:1527/playerDB` connection under the Databases node.

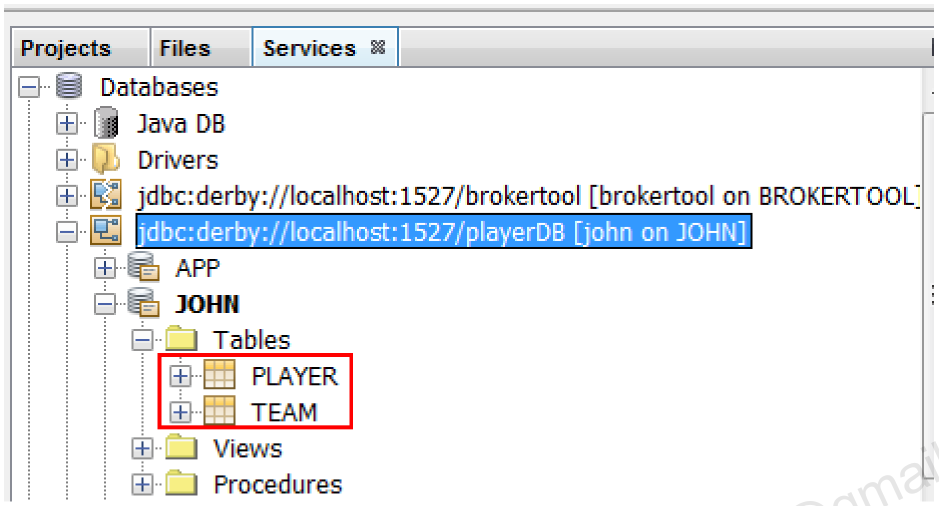
b. Right-click the connection and select Refresh.

c. Expand the JOHN schema. You see the nodes for the tables, views, and procedures.

3. Expand the Tables node to see the PLAYER and TEAM tables.

   Right-click the PLAYER table node and select View Data. A SQL command window opens and executes an SQL command to display the data in the table.

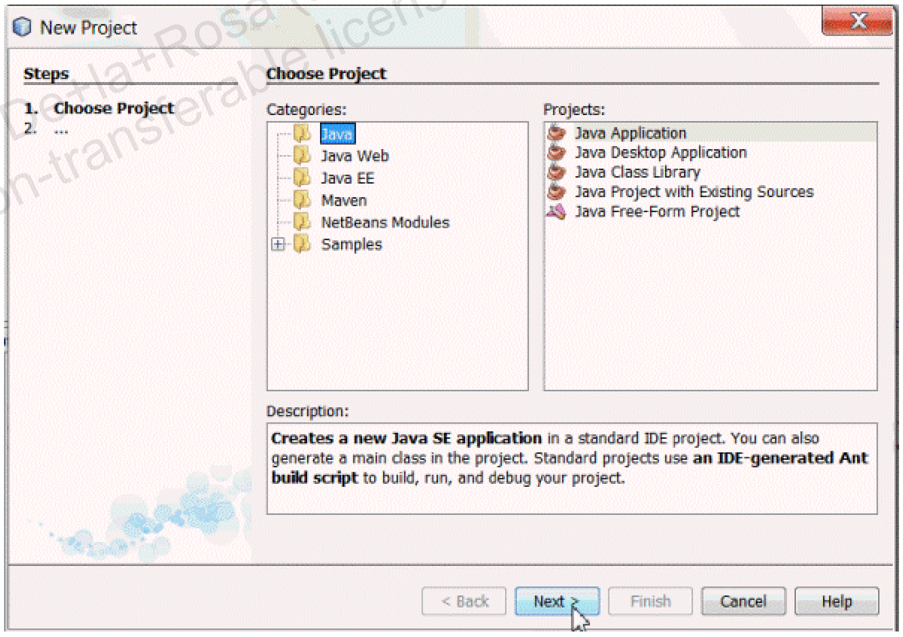
   a. Repeat the previous step for the TEAM table.


Projects
Files
Services
Databases
Java DB
Drivers
jdbc:derby://localhost:1527/brokertool [brokertool on BROKERTOOL]
jdbc:derby://localhost:1527/playerDB [john on JOHN]
APP
JOHN
Tables
PLAYER
TEAM
Views
Procedures

4. Generating Entity classes from the database.

   a. To create a new Java project, select File > New Project from within NetBeans.

   b. Select Java from the Categories column and Java Application from the Projects column. Then click Next.



Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

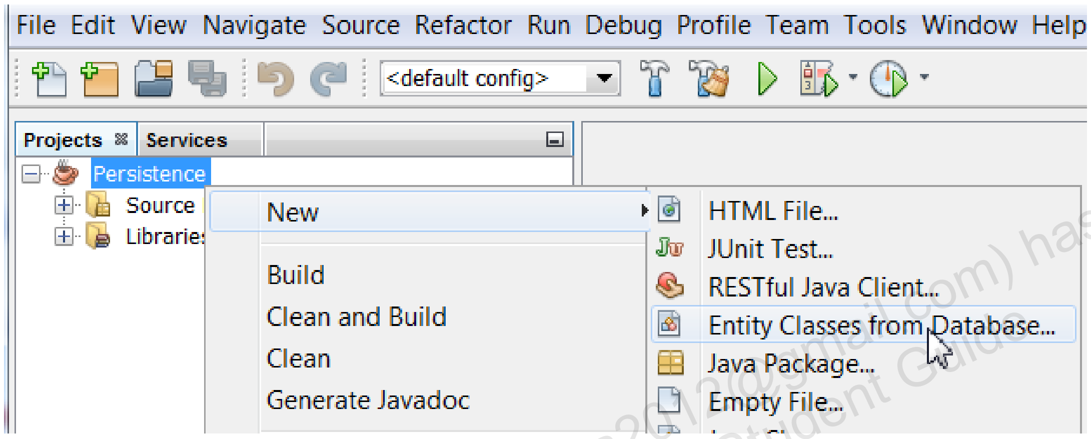
c. Perform the following steps:

   1) Name the project `Persistence`.

   2) Provide the Project Location as `D:\labs\09-IntroJPA\practices\practice01`. Deselect the Create Main Class check box.

   3) Click Finish.

d. Right-click the `Persistence` project and select New > "Entity Classes from Database."



e. Enter the following information to create Entity classes:

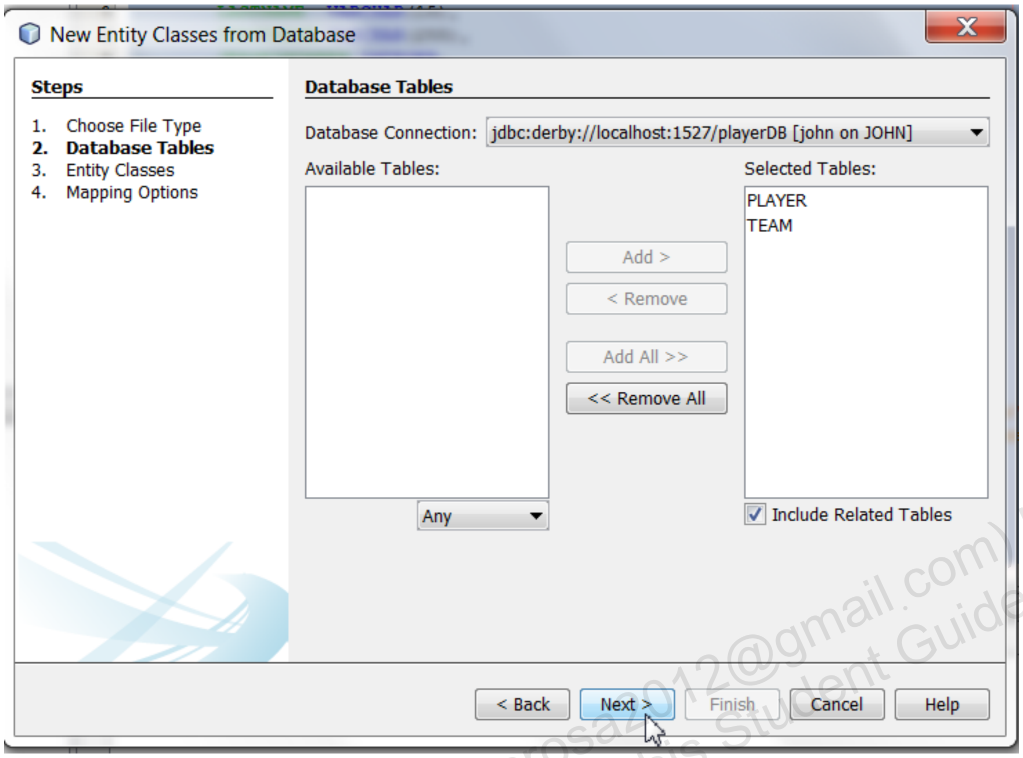
   1) In the Database Connection field, select `jdbc:derby://localhost:1527/playerDB` from the drop-down list.

   2) You see the PLAYER and TEAM tables in the Available Table category

   3) Click Add All.

   4) You see the PLAYER and TEAM tables in the Selected Table category.

5) Click Next.



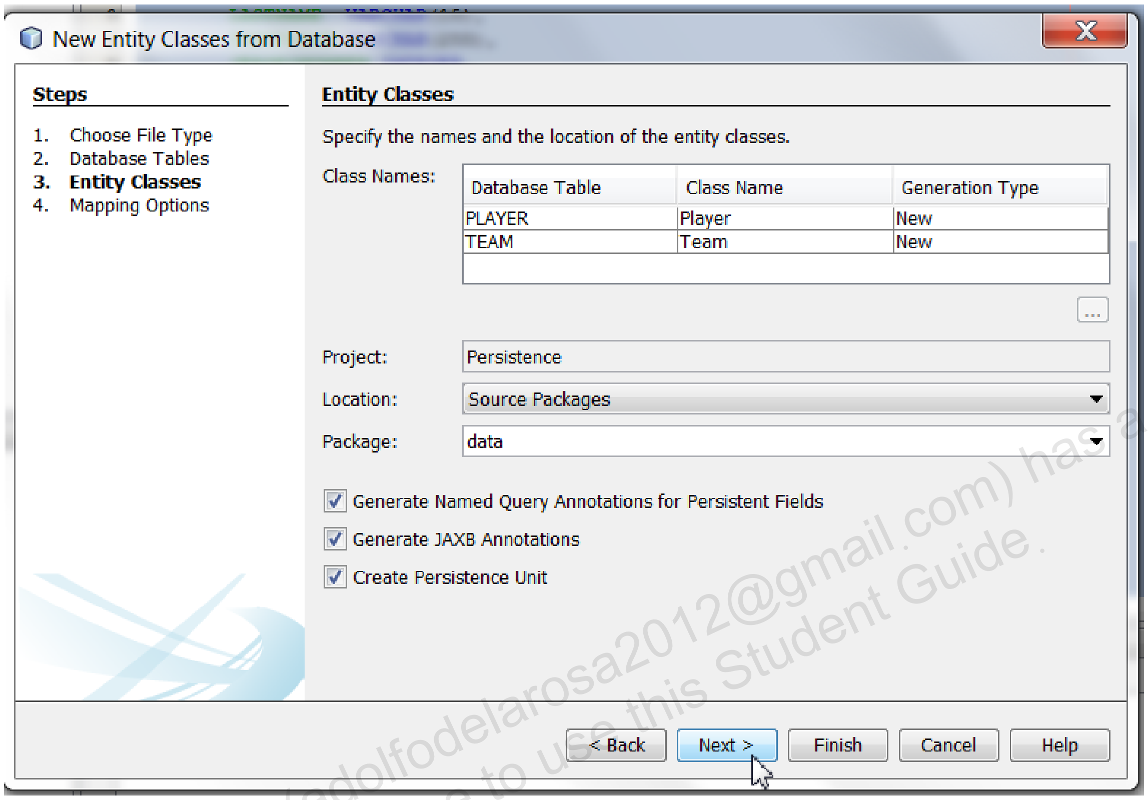
Database Connection: jdbc:derby://localhost:1527/playerDB [john on JOHN]

Available Tables:

Selected Tables:
PLAYER
TEAM

[✓] Include Related Tables

5. In the Entity Classes window, enter `data` as the package name and click Next.



Entity Classes

Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Generation Type
PLAYER	Player	New
TEAM	Team	New

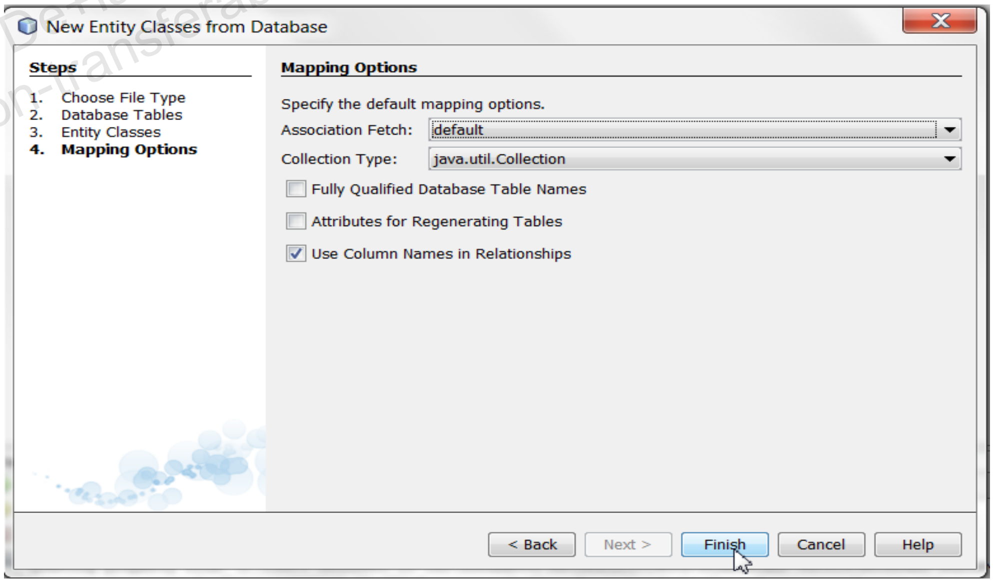
Project: Persistence

Location: Source Packages

Package: data

- [✓] Generate Named Query Annotations for Persistent Fields
- [✓] Generate JAXB Annotations
- [✓] Create Persistence Unit

6. In the Mapping Options window, click Finish with the default selections.



Mapping Options

Specify the default mapping options.

Association Fetch: default

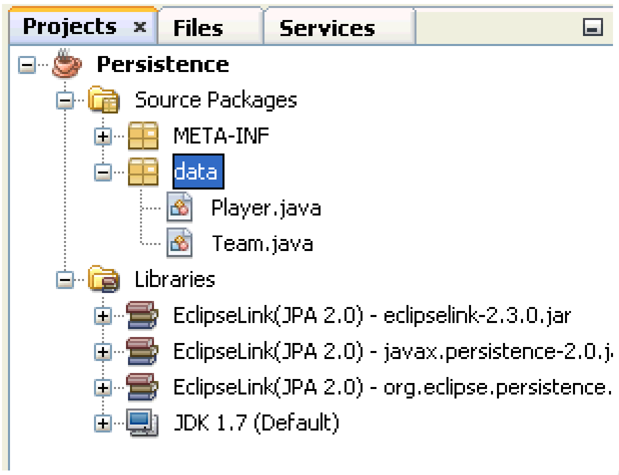
Collection Type: java.util.Collection

- [ ] Fully Qualified Database Table Names
- [ ] Attributes for Regenerating Tables
- [✓] Use Column Names in Relationships

7. Verify the creation of Entity classes.

   a. Select the `Persistence` project.

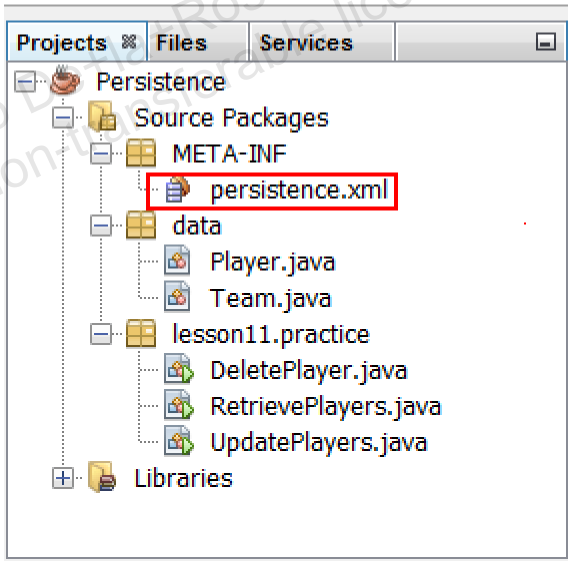
   b. Open the `data` package, where you should see `Team.java` and `Player.java` created.


Projects
Files
Services
Persistence
Source Packages
META-INF
data
Player.java
Team.java
Libraries
EclipseLink(JPA 2.0) - eclipselink-2.3.0.jar
EclipseLink(JPA 2.0) - javax.persistence-2.0.j
EclipseLink(JPA 2.0) - org.eclipse.persistence.
JDK 1.7 (Default)

The set of entities created in the application is called a *persistence unit*. An application's persistence unit is defined in a configuration file called `persistence.xml`. This file exists in the `META-INF` directory.

8. Name the persistence unit `PersistencePU` in the `persistence.xml` file.

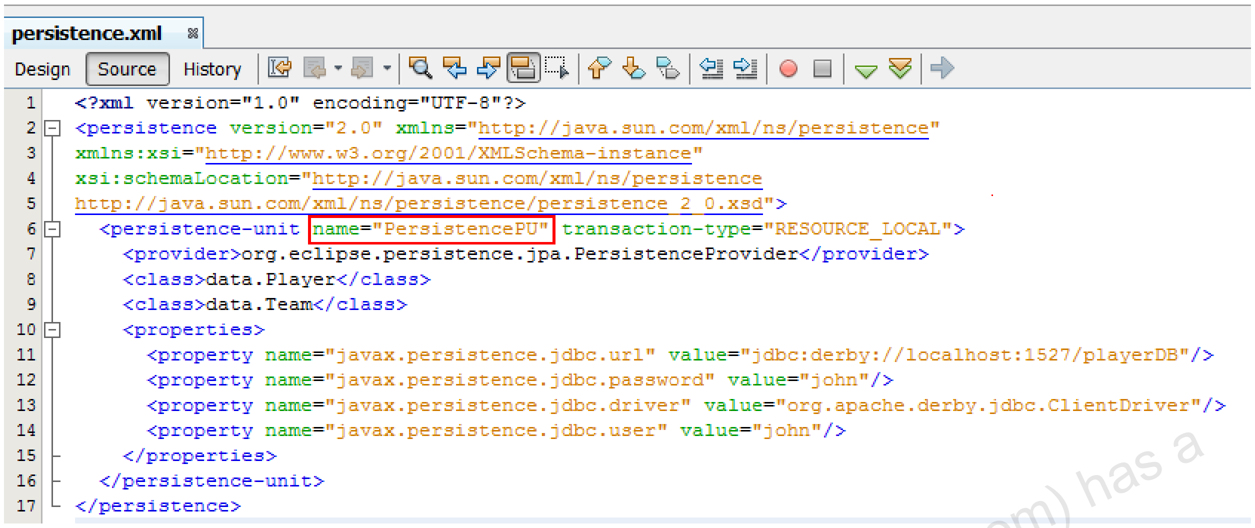
   a. Expand the `META-INF` folder.



   b. Click `persistence.xml` to open it in the code editor window.

   c. Click the Source tab.

d. Verify that the name of the persistence unit is `PersistencePU` as follows:



```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="PersistencePU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>data.Player</class>
    <class>data.Team</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/playerDB"/>
      <property name="javax.persistence.jdbc.password" value="john"/>
      <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.user" value="john"/>
    </properties>
  </persistence-unit>
</persistence>
```

To resolve any reference problem, add the `derbyclient.jar` file to the project library located in the `./09-IntroJPA/resources` folder. You may find the file in your JDK installation location as well.

Practice 9-2: (Summary Level) Implementing CRUD Operations by Using JPA

Overview

In this practice, you learn about the following to implement CRUD operations:

- Important packages and classes of the API
- How to use the API in a sample application
- How to use JPQL

Assumptions

You should have completed Practice 9-1.

Tasks

1. In NetBeans, complete the following steps to open the `Persistence` project:
   a. Select File > Open Project.
   b. In the Open Project dialog box, browse to the `D:\labs\09_IntroJPA\practices\practice02` folder.
   c. Select the `Persistence` project and click the Open Project button.
2. In the Projects window, expand `Persistence > Source Packages > data`.
3. Double-click `Player.java` to open it in the code editor window.
4. Make the following change to the `Player.java` Entity class:
5. Override the `toString()` method.

```
  @Override
  public String toString() {
 String player = String.format("[Jersey Number: %d, Name: %s,ln: %s]",
jerseynumber, firstname, lastname);
 return player;
}
```

6. To insert records in the `Player` table, create a java class named `CreatePlayers.java` in the `lesson09.practice` package.
   a. Import the `javax.persistence` package's classes to `CreatePlayers.java`.
   b. Create a `main()` method and add code to insert two records in the Player table. This operation should be within a transaction.
   c. Add the Derby driver to connect to the Derby database server.
      1) Select Project > Properties and right-click Libraries.
      2) Select Add Jar/Folder.
      3) Browse to `D:\labs\09-IntroJPA\resources\derbyclient.jar`.
      4) Click Open, and then Click OK.
   d. Run `CreatePlayers.java`.

e. Examine the database.

   1) Select JOHN schema > Expand Tables Node >Player Table.

   2) Right-click the Player table node and select View Data. You see the two rows inserted in the table.

7. Complete the following steps to retrieve the data from the `Players` table:

   a. Expand the `Persistence` project and select `RetrievePlayer.java` from the `lesson09.practice` package.

   b. Add code to the `main()` method of `RetrievePlayer.java` to retrieve all players and print them in the console.

   c. Run `RetrievePlayers.java` and select Run File.

   d. Verify the output. You see all the records in the `Player` table displayed in the console.

8. Complete the following steps to update a specific record in the `Player` table.

   a. Expand the `Persistence` project and select `UpdatePlayers.java` from the `lesson09.practice` package.

   b. Add code in the `main()` method to set the `jerseyNumber` to 60 for the player whose `PlayerId` is 5.

   c. Run `UpdatePlayers.java`.

   d. Verify the output by examining the contents of the Player table in the `playerDB` database.

9. Complete the following steps to delete a specific record in the `Player` table:

   a. Expand the `Persistence` project and select `DeletePlayer.java` from the `lesson09.practice` package.

   b. Double-click `DeletePlayer.java` in the Project pane to open it in the code editor window.

   c. Add code to the `main()` method that deletes a Player whose `PlayerId` is 5.

   d. Run `DeletePlayer.java`.

   e. Examine the contents of the Player table in the `PlayerDB` database and verify that the record is deleted.

To resolve any reference problems, add the `derbyclient.jar` file to the project library located in the `./09-IntroJPA/resources` folder. You may find the file in your JDK installation location as well.

Practice 9-2: (Detail Level) Implementing CRUD Operations by Using JPA

Overview

In this practice, you learn about the following to implement CRUD operations:

- Important packages and classes of the API
- How to use the API in a sample application
- How to use JPQL

Assumptions

You have completed Practice 9-1.

Tasks

1. In NetBeans, complete the following steps to open the `Persistence` project:
   a. Select File > Open Project.
   b. In the Open Project dialog box, browse to the `D:\labs\09_IntroJPA\practices\practice02` folder.
   c. Select the `Persistence` project and click the Open Project button.
2. In the Projects window, expand `Persistence > Source Packages > data`.
3. Double-click `Player.java` to open it in the code editor window.
4. Make the following change to the `Player.java` Entity class:
5. Override the `toString()` method.

```
  @Override
  public String toString() {
 String player = String.format("[Jersey Number: %d, Name: %s,ln: %s]",
jerseynumber, firstname, lastname);
 return player;
    }
```

6. Complete the following steps to insert records in the Player table:
   a. Right-click the `Persistence` project and select New > Java Class.
   b. Save the class as `CreatePlayers` and select `lesson09.practice` as the package name. Click Finish.
   c. Click `CreatePlayers.java` in the Project pane to open it in the code editor window.
   d. Import the following classes:

```
import data.Player;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
```

e. Create a `main` method in the class and add the following lines of code to create an `EntityManager` instance using the `EntityManagerFactory` class.

```
public static void main(String[] args) {
 EntityManagerFactory emf =
Persistence.createEntityManagerFactory("PersistencePU");
 EntityManager em = emf.createEntityManager();
}
```

f. Add the following code to the `main` method.

```
         em.getTransaction().begin();

        Player p1 = new Player();
        p1.setId(5);
        p1.setFirstname("Enid");
        p1.setJerseynumber(30);
        p1.setLastname("Blyton");
        p1.setLastspokenwords("I am in the best form");
        em.persist(p1);

        Player p2 = new Player();
        p2.setId(6);
        p2.setFirstname("Mark");
        p2.setJerseynumber(40);
        p2.setLastname("Twain");
        p2.setLastspokenwords("I will be back");
        em.persist(p2);

       em.getTransaction().commit();

       em.close();
       emf.close();
```

The preceding code creates a transaction and two objects of the `Player` class that are persisted as two rows in the `Player` table.

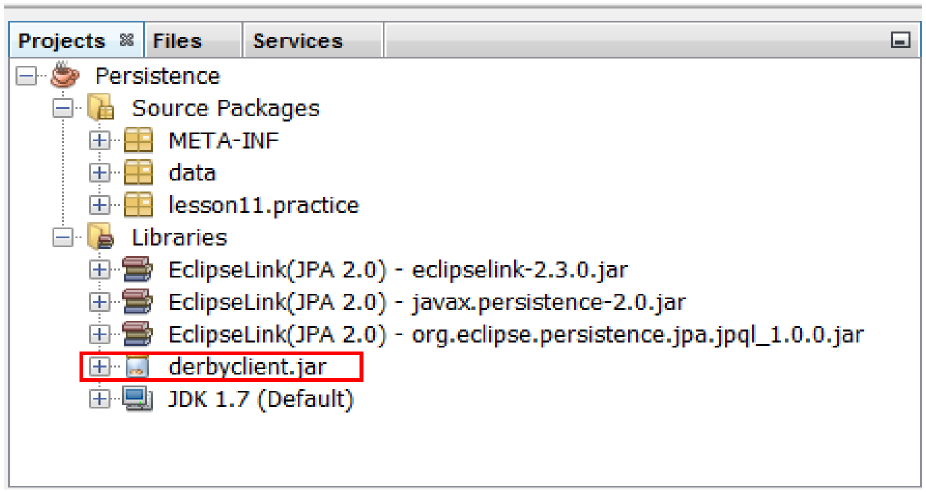
g. Add the Derby driver to connect to the Derby database server.

1) Select Project > Properties and right-click Libraries.

2) Select Add Jar/Folder.

3) Browse to `D:\labs\09-IntroJPA\resources\derbyclient.jar`.

4) Click Open, and then click OK.



```
Persistence
  Source Packages
    META-INF
    data
    lesson11.practice
  Libraries
    EclipseLink(JPA 2.0) - eclipselink-2.3.0.jar
    EclipseLink(JPA 2.0) - javax.persistence-2.0.jar
    EclipseLink(JPA 2.0) - org.eclipse.persistence.jpa.jpql_1.0.0.jar
    derbyclient.jar
    JDK 1.7 (Default)
```

h. In the Projects window, right-click `CreatePlayers.java` and select Run File.

i. Examine the contents of the database.

1) In the Services window, expand the `jdbc:derby://localhost:1527/playerDB` connection under the Databases node.

2) Right-click the connection and select Refresh.

3) Expand JOHN schema > Expand Tables Node > Player Table.

4) Right-click the `Player` table node and select View Data. You see the two rows inserted in the table.

7. Complete the following steps to retrieve the data from the `Players` table:

a. Expand the `Persistence` project and select `RetrievePlayer.java` from the `lesson09.practice` package.

b. Double-click `RetrievePlayers.java` in the Project pane to open it in the code editor window.

c. Add the following code to the `main()` method.

```
 Query q = em.createQuery("select c from Player c");

        List<Player> playerList = q.getResultList();

        for (Player p : playerList) {
            System.out.println(p.toString());
        }
```

d. In the Projects window, right-click `RetrievePlayers.java` and select Run File.

e. Verify the output. You see all the records in the `Player` table displayed in the console.

8. Complete the following steps to update a specific record in the `Player` table:
   a. Expand the `Persistence` project and select `UpdatePlayers.java` from the `lesson09.practice` package.
   b. Double-click `UpdatePlayers.java` in the Project pane to open it in the code editor window.
   c. Add the following code to update a row in `Player.java`.
   d. Add the following code to the `main()` method.

```
em.getTransaction().begin();
Query q = em.createQuery("update Player p set p.jerseynumber =
60 where p.id= 5");
q.executeUpdate();
em.getTransaction().commit();
```

The preceding code searches for the record in the `Player` table with `playerId` equal to 5.

   e. In the Projects window, right-click `UpdatePlayers.java` and select Run File.
   f. Examine the contents of the database.
      1) In the Services window, expand the `jdbc:derby://localhost:1527/playerDB` connection under the Databases node.
      2) Right-click the connection and select Refresh.
      3) Expand JOHN schema > Expand Tables Node > Player Table.
      4) Right-click the `Player table` node and select View Data. You see that the `jerseynumber` in the row with the `playerID` equal to 5 has been updated in the table.

9. Perform the following steps to delete a specific record in the `Player` table:
   a. Expand the `Persistence` project and select `DeletePlayer.java` from the `lesson09.practice` package.
   b. Double-click `DeletePlayer.java` in the Project pane to open it in the code editor window.
   c. Add the following code to the `main` method to search for a `playerID` of 5:

```
 em.getTransaction().begin();
 Player player = em.find(Player.class, 5);
        if (player != null) {
            System.out.println(player.toString());
            em.remove(player);
           }
em.getTransaction().commit();
```

   d. In the Projects window, right-click `DeletePlayer.java` and select Run File.

e. Examine the contents of the database:

   1) In the Services window, expand the `jdbc:derby://localhost:1527/playerDB` connection under the Databases node.
   2) Right-click the connection and select Refresh.
   3) Expand JOHN schema > Expand Tables Node > Player Table.
   4) Right-click the `Player` table node and select View Data. You see that the row in which the `playerID` equals 5 has been deleted from the table.

To resolve any reference problems, add the `derbyclient.jar` file to the project library located in the `./09-IntroJPA/resources` folder. You may find the file in your JDK installation location as well.

Practices for Lesson 10: Applying the JPA

Chapter 10

Practices for Lesson 10

Practices Overview

In these practices, you will apply the JPA in a two-tier application. You will:

- Identify entity relationships in the BrokerTool application
- Implement database connectivity in the BrokerTool application by using the JPA

Practice 10-1: (Summary Level) Identifying Entity Relationships in the BrokerTool Application

Overview

In this practice, you will create the BrokerTool database, generate entities, and analyze the generated code. Java DB database server is part of NetBeans. We will use Java DB as the database server for the BrokerTool application.

Assumptions

NetBeans is running. ***You can skip Tasks #2 and 3, if you have completed the practices for Lesson 2.***

Tasks

1. Ensure that Java DB is running.
2. Create the BrokerTool database.
   a. Enter the following values in the Create JavaDB Database dialog box:
      1) Database Name: `BrokerTool`
      2) User Name: `brokertool`
      3) Password: `brokertool`
   b. Validate that the database and a connection have been created under the Java DB node and Databases node respectively. The Connection name should be: jdbc`:derby://localhost:1527/brokertool [brokertool on BROKERTOOL]`
3. Run the SQL script file `brokertool.sql` located at: `D:\labs\10-ApplyJPA\practices\practice01` to create tables and populate them with data in the BrokerTool database.
   — In the services window, refresh the BrokerTool connection, expand it, and verify that the tables have been created and populated with data from the BrokerTool schema.
4. Define persistent entities in the BrokerTool application.
   a. In NetBeans, open the `BrokerTool_2Tier` project located in the `D:\labs\10-ApplyJPA\practices\practice01` folder. This project will show some compilation errors that can be ignored at this point.
   b. Add the following libraries to the `BrokerTool_2Tier` project:
      1) EclipseLink (JPA 2.0)
      2) `Derbyclient.jar` located at `D:\Program Files\JDK1.7.x\db\lib` (The file can also be obtained from the `D:\labs\resources\setup` folder.)
   c. In the Projects window, create a new Java package and name it `brokermodel` if it does not already exist.
   d. Create entity classes from the BrokerTool database. Include all the tables.
5. Examine the generated entity classes and identify the various annotations, such as `@Entity`, `@Table`, and `@NamedQueries`.

6. Verify that `persistence.xml` file has been generated in the META-INF node of source packages.
   a. Identify and note the following:
      1) Persistent Unit Name
      2) Persistent Library
      3) JDBC Connection
   b. Ensure that transaction type is “`RESOURCE_LOCAL`” and the JDBC properties are correctly listed.
7. Save the changes and keep the project open in NetBeans.

Practice 10-1: (Detailed Level) Identifying Entity Relationships in the BrokerTool Application

Overview

In this practice, you will create the BrokerTool database, generate entities, and analyze the generated code. Java DB database server is part of NetBeans. We will use Java DB as the database server for the BrokerTool application.

Assumptions

NetBeans is running. ***You can skip Tasks #2 and 3, if you have completed the practices for Lesson 2.***

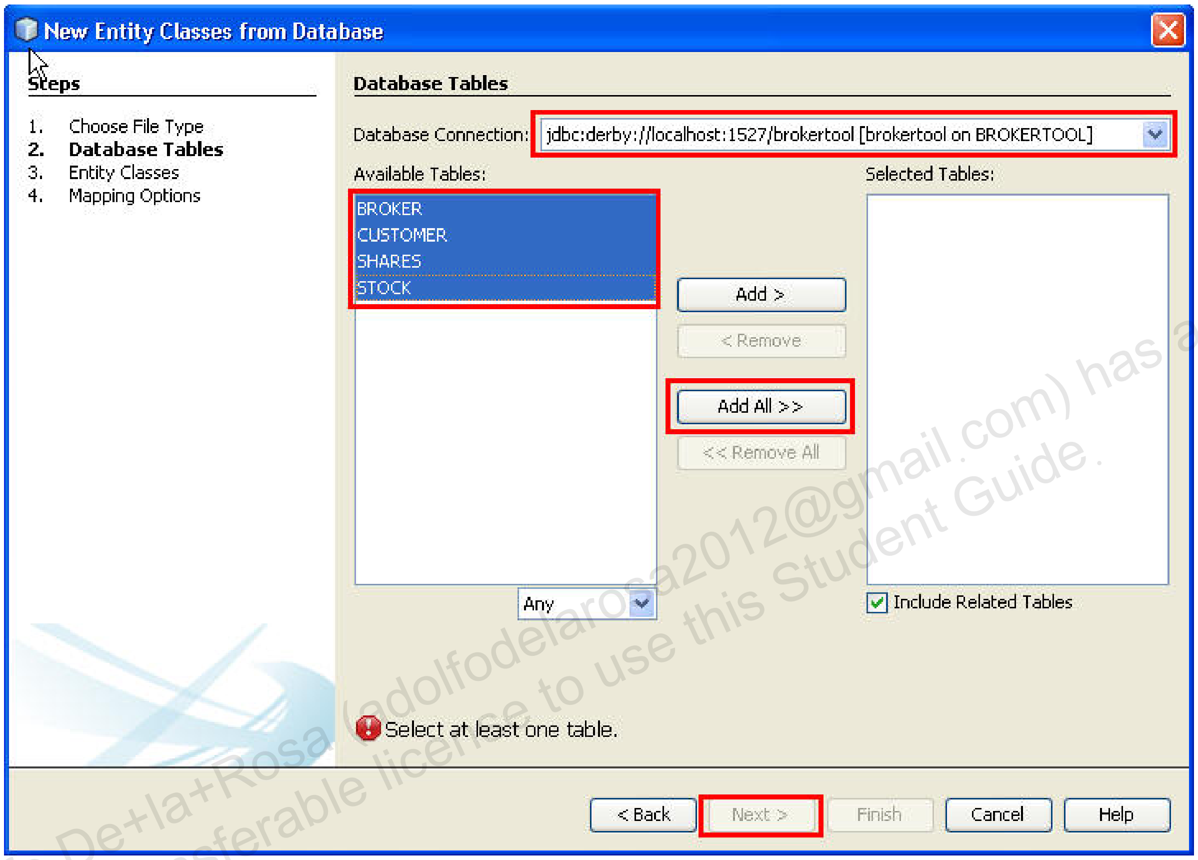
Tasks

1. Ensure that Java DB is running.
   a. In NetBeans, switch to the Services window.
   b. Right-click the JavaDB node and start the server if it is not started already.
2. Create the BrokerTool database.
   a. In the Services window, right-click JavaDB and select Create Database.
   b. Enter the following values in the Create JavaDB Database dialog box:
      1) Database Name: `BrokerTool`
      2) User Name: `brokertool`
      3) Password: `brokertool`
   c. Validate that the database and a connection have been created under the Java DB node and Databases node respectively. The Connection name should be: jdbc`:derby://localhost:1527/brokertool [brokertool on BROKERTOOL]`
3. Run a SQL script to create tables and populate them with data from the BrokerTool database.
   a. In NetBeans, open the `brokertool.sql` file located at: `D:\labs\10-ApplyJPA\practices\practice01`
   b. In the SQL command window, ensure that the BrokerTool connection is selected in the drop-down list.
   c. Execute the SQL scripts.
   d. In the services window, refresh the Brokertool connection, expand it, and verify that the tables have been created in the BrokerTool schema.
   e. Right-click the Shares table and select View Data. Examine and verify the table records in the SQL commands output window.
   f. Repeat the previous step for the Broker, Stock, and Customer tables.
4. Define persistent entities in the BrokerTool application.
   a. In NetBeans, open the `BrokerTool_2Tier` project located in the `D:\labs\10-ApplyJPA\practices\practice01` folder. This project will show some compilation errors that can be ignored at this point.
   b. Add the following libraries to the `BrokerTool_2Tier` project:
      1) EclipseLink (JPA 2.0)
      2) `Derbyclient.jar` located at `D:\Program Files\JDK1.7.x\db\lib` (The file can also be obtained from the `D:\labs\resources\setup` folder.)

c. In the Projects window, create a new Java package and name it `brokermodel` if it does not already exist.

d. Right-click `brokermodel` package and select New > “Entity Classes from Database.” Provide information in the ensuing dialog boxes as shown in the following figures.

   1) Select all the tables, click Add All >> and then click Next.



Database Connection: jdbc:derby://localhost:1527/brokertool [brokertool on BROKERTOOL]

Available Tables:

BROKER
CUSTOMER
SHARES
STOCK

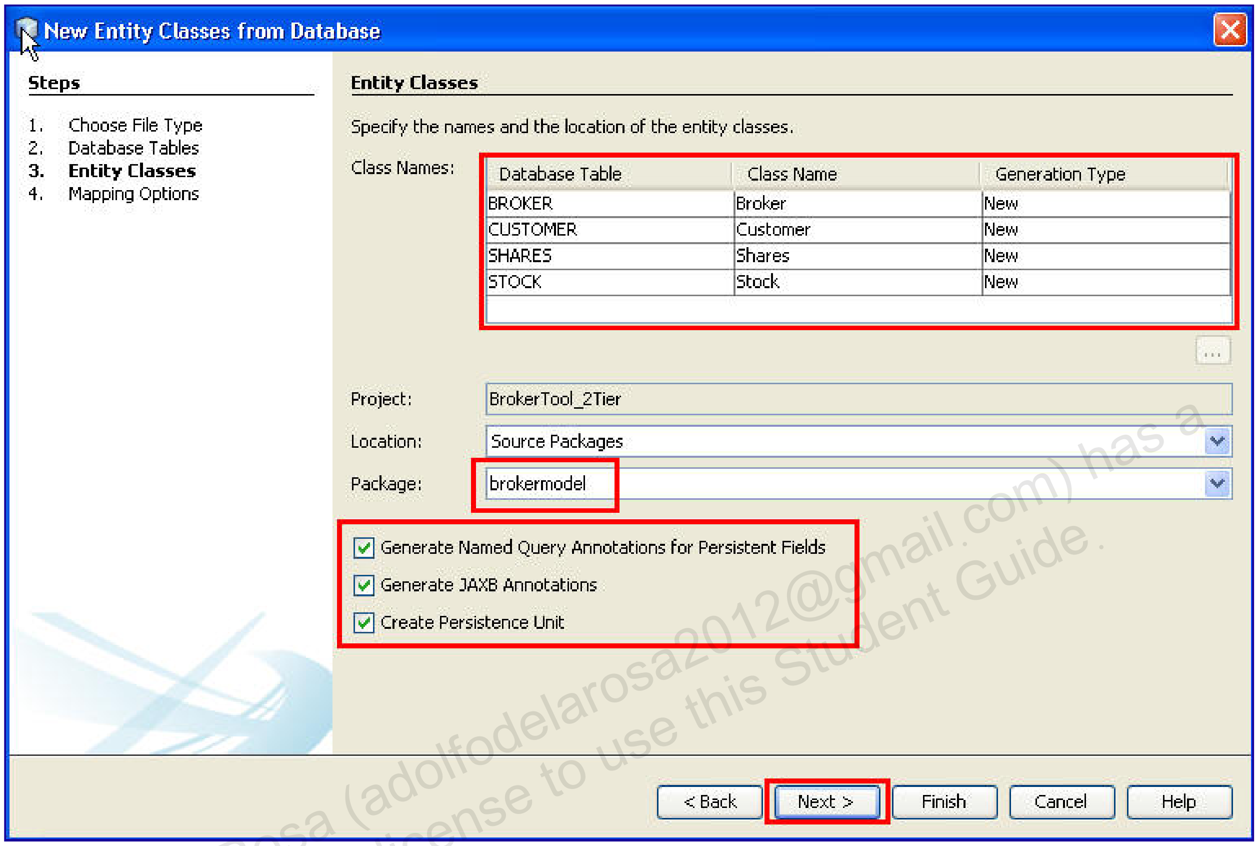
Any

Selected Tables:

[✓] Include Related Tables

Select at least one table.

2) Review the class name and the generation type of the entity classes. Select `brokermodel` as the package. Select the two check boxes (all three options with check boxes should be selected), and click Next.



New Entity Classes from Database

Entity Classes

Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Generation Type
BROKER	Broker	New
CUSTOMER	Customer	New
SHARES	Shares	New
STOCK	Stock	New

Project: BrokerTool_2Tier

Location: Source Packages

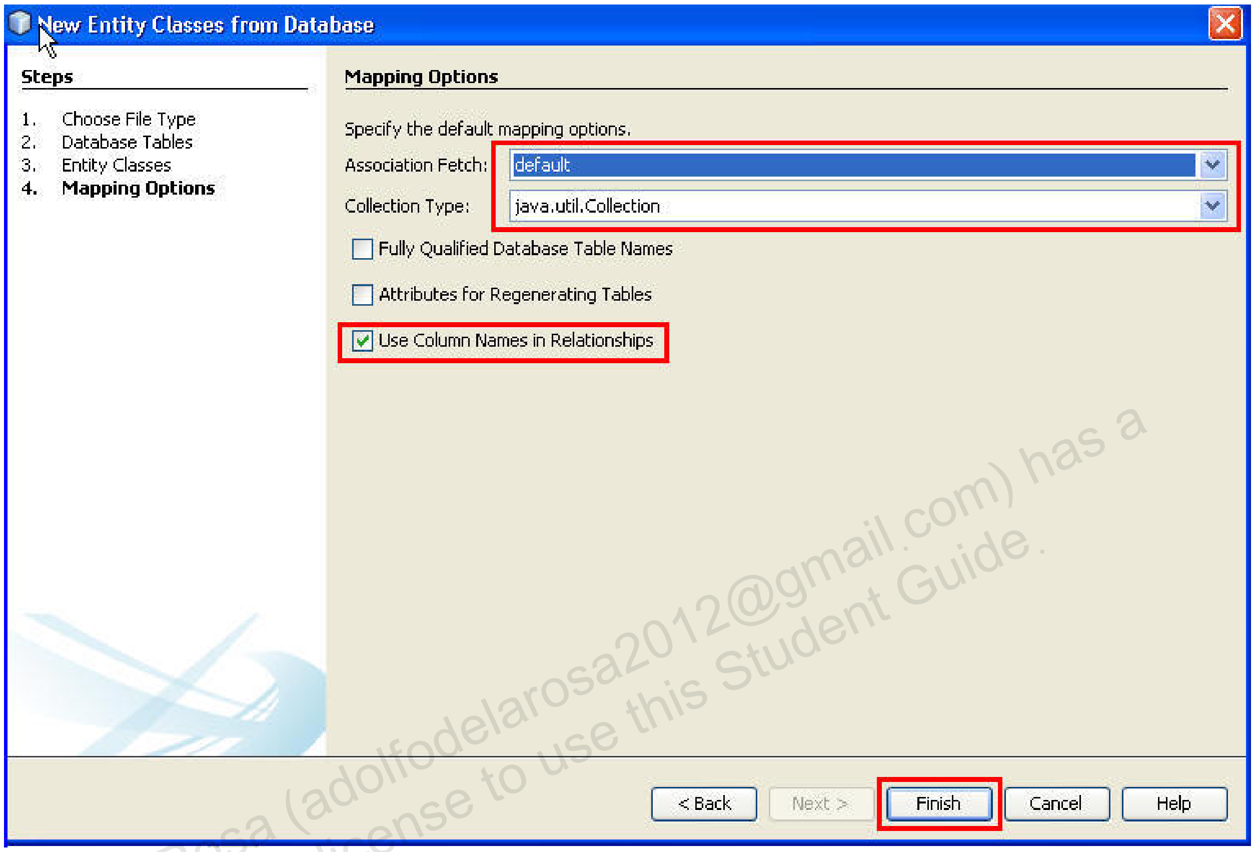
Package: brokermodel

✓ Generate Named Query Annotations for Persistent Fields

✓ Generate JAXB Annotations

✓ Create Persistence Unit

3) Select the values for the Association Fetch and Collection Type fields as shown in the following screenshot. Then select the “Use Column Names in Relationships” option and click the Finish button.



New Entity Classes from Database

Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. **Mapping Options**

Mapping Options

Specify the default mapping options.

Association Fetch: default

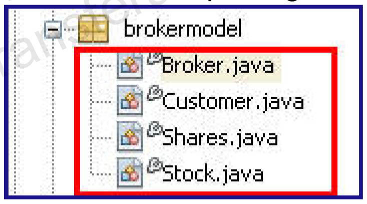
Collection Type: java.util.Collection

☐ Fully Qualified Database Table Names

☐ Attributes for Regenerating Tables

☑ Use Column Names in Relationships

4) In the Projects window, verify that the following classes are generated in the `brokermodel` package.



```
brokermodel
  Broker.java
  Customer.java
  Shares.java
  Stock.java
```

5. Examine the generated entity classes.
   a. In the Projects window, expand the `brokermodel` package under the `Source Packages` node.
   b. Click `Broker.java` to open it in the code editor window.
   c. Examine the code and identify the various annotations, such as `@Entity`, `@Table`, and `@NamedQueries`.
   d. Identify the annotation and its attributes. These denote the relationship between the Broker and Customer tables.
   e. Repeat sub-step **d** for the `Customer`, `Shares`, and `Stock` entity classes.

6. In the projects window, expand the META-INF node under Source Packages.
   a. Verify that `persistence.xml` file has been generated.
   b. Click `persistence.xml` to view it in the code editor window.
   c. Identify and note the following:
      1) Persistent Unit Name
      2) Persistent Library
      3) JDBC Connection
      4) Entity classes
   d. Click the source tab and verify that the transaction type is “`RESOURCE_LOCAL`” as shown below. This ensures that JPA can be used outside a Java EE environment.
```
<persistence-unit name="BrokerTool_2TierPU" transaction-
type="RESOURCE_LOCAL">
```
   e. The JDBC properties are correctly listed:
```
<properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:1527/brokertool"/>
      <property name="javax.persistence.jdbc.password"
value="brokertool"/>
      <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.user"
value="brokertool"/>
    </properties>
```
7. Save the changes and keep the project open in NetBeans for the next practice.

Practice 10-2: (Summary Level) Implementing Database Connectivity in the BrokerTool Application by Using the JPA

Overview

In this practice, you will use JPA in the BrokerTool _2Tier application. You will create the DAO classes and use them in the JavaFX classes.

- When the BrokerTool_2Tier application runs, the broker dashboard opens with the shares list of the customer with customer_id=1.
- When a user selects a particular customer from the customer choice box, the dashboard reflects the shares information of that customer.
- The piechart also gets updated accordingly.

Assumptions

You have completed the previous practice successfully.

Tasks

1. In the Projects window, expand the `BrokerTool_2Tier` project and verify that `persistence.xml` exists and `Broker`, `Shares`, `Customer`, and `Stock` entity classes exist.
2. Create the `BrokerEntityManager` class.
   a. Under Source Packages, create a new package and name it `BrokerToolDAO`.
   b. Create the `BrokerEntityManager` class in `BrokerToolDAO` package.
      Declare instances of the `EntityManager` and `EntityManagerFactory` classes.
   - Write a create method to initialize the EntityManagerFactory instance. Use the EntityManagerFactory instance to create and initialize an EntityManager instance.
   - Write a close method to close the instances created in the previous sub-step.

   c. Create the required DAO classes.
      1) Create an abstract class and name it `AbstractBrokerDAO`.
      2) Refer to the `AbstractBrokerDAO.java` file in `D:\labs\10-ApplyJPA\practices\practice02\brokertoolDAO` and complete the `create`, `find`, `findAll`, `findRange`, `edit`, and `delete` methods.
      3) Create `CustomerDAO`, `BrokerDAO`, `SharesDAO`, and `StockDAO` classes such that they extend from the `AbstractBrokerDAO` class.
      4) In each of the classes, override the parent class methods.
      5) Invoke the super class' method and pass the corresponding entity class as a parameter. For example, the `create` method of `CustomerDAO` class should look like the following code snippet:

```
public void create(Customer entity) {
        super.create(entity);
```

   d. At the end of this task, your DAO classes and entity classes are ready to be used by the JavaFX classes.

3. In this and the following tasks, you will modify the front end of the JavaFX classes. Complete the following steps to examine the `BrokerDashboard.fxml`.
   a. Expand the `brokertoolclient` package.
   b. Click `BrokerDashboard.fxml` to open it in the editor window.
   c. Identify the controller for the file.
   d. Identify the event handler for the customer choice box.
   e. Identify the IDs for the piechart and table.
4. Complete the following steps to modify `BrokerDashboard.java`, which is the controller of `BrokerDashboard.fxml`.
   a. Verify that the choice box, tableview, and piechart JavaFX UI elements are declared.
   b. Declare the `SharesDAO`, `CustomerDAO`, and `Customer` instances and make the required imports.
   c. Modify the `brokerdashboard` constructor to initialize the `CustomerDAO` instance.
   d. Create an event handler for the customer choice box such that the event listener method fetches sales data for the selected customer. Refer to the following code:

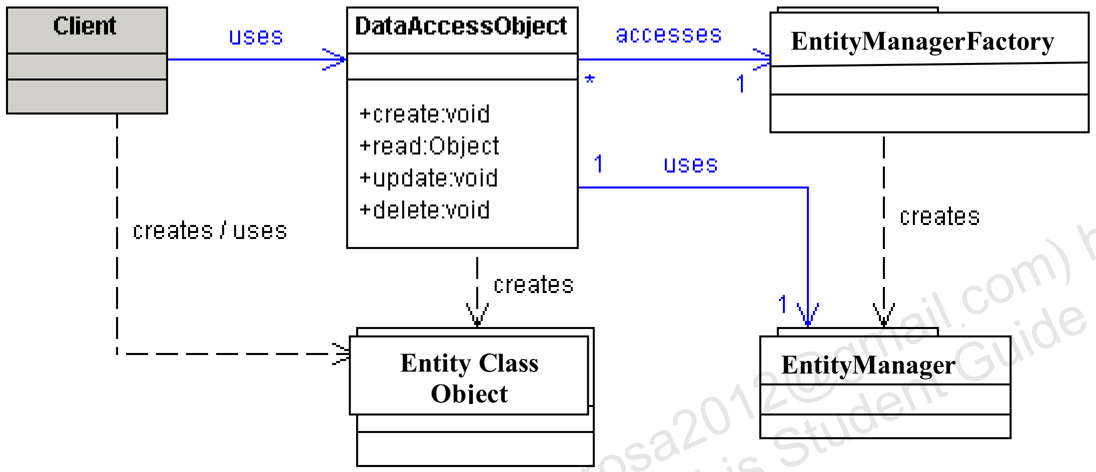
```
customerChoiceBox.getSelectionModel().selectedIndexProperty().ad
dListener(new ChangeListener<Number>(){
            @Override
       public void changed(ObservableValue <? extends Number>
ov, Number value, Number newValue){
            if (newValue.intValue()>0 ){

sharesService.setUserAndPw(BrokerToolClientApp.getInstance().get
User(),BrokerToolClientApp.getInstance().getPassword(),
customerChoiceBox.getSelectionModel().getSelectedIndex()+1);
                if (sharesService.isRunning())
           sharesService.createTask();
             else
             sharesService.restart();
}
        }

  });
```

   e. Review the `GetSharesService.java` and `GetSharesTask.java` classes to figure out how the customer's shared data is retrieved from the database.
   f. Examine the `call` method of `GetSharesTask.java`, which is equivalent to the run method of a Java thread.
5. Save the changes in the `BrokerDashboard.java` file, and build and run the project. Ensure that the JavaDB server is started.
   a. Verify the broker dashboard for sales data of customer 1.
   b. Select the third customer from the customer choice box and verify that the records of that customer are displayed in the broker dashboard.

Practice 10-2: (Detailed Level) Implementing Database Connectivity in the BrokerTool Application by Using the JPA

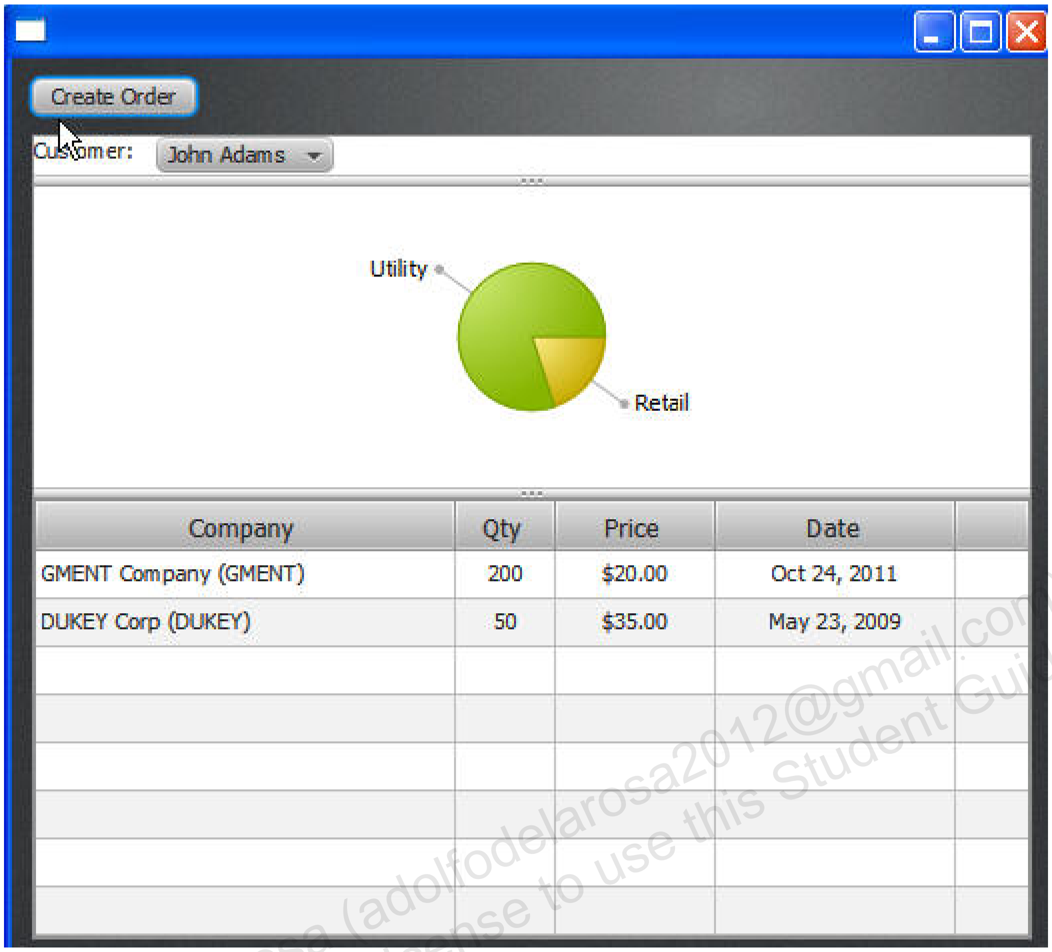
Overview

In this practice, you will use JPA in the BrokerTool _2Tier application. The following diagrams depict the class diagram of the BrokerTool two-tier application.



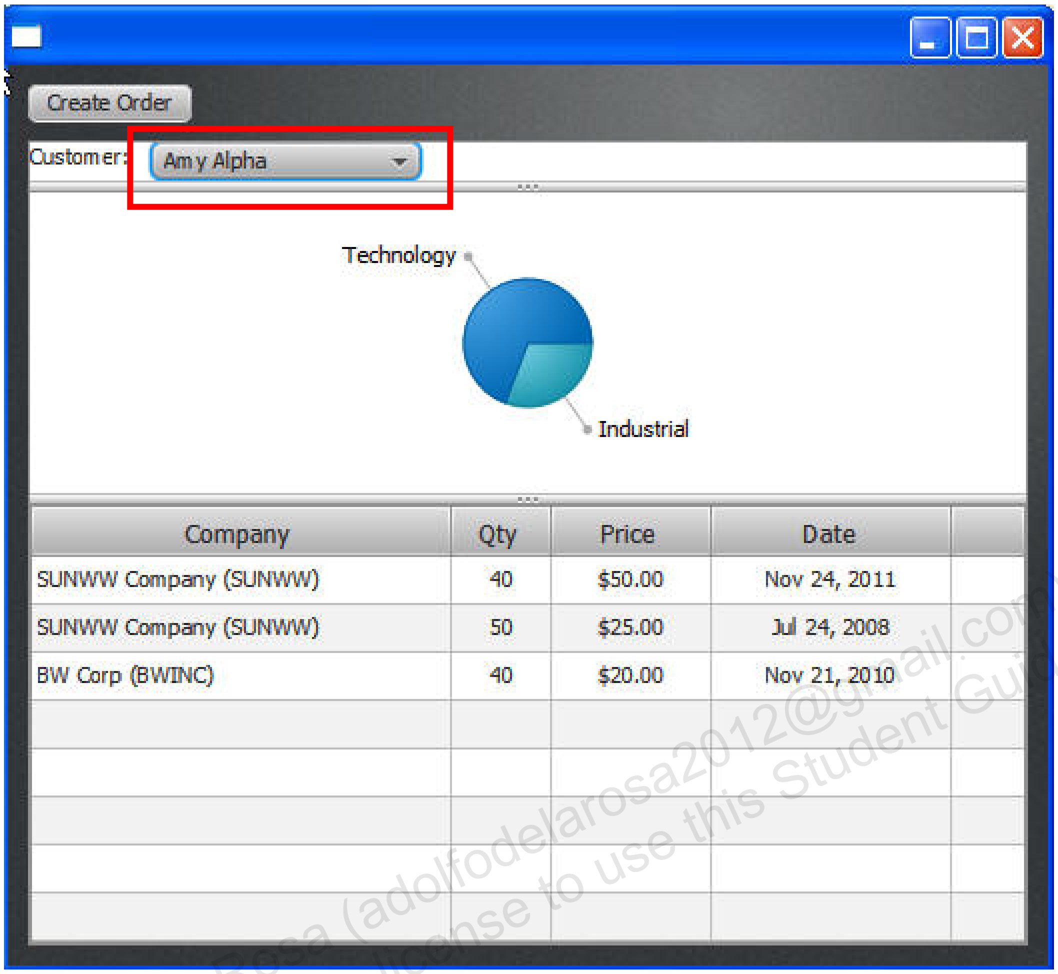
In this practice, you will create the DAO classes and use them in the JavaFX classes.

- When the BrokerTool_2Tier application runs, the broker dashboard opens with the shares list of the customer with customer_id=1.



Company	Qty	Price	Date	
GMENT Company (GMENT)	200	$20.00	Oct 24, 2011	
DUKEY Corp (DUKEY)	50	$35.00	May 23, 2009	

- When a user selects a particular customer from the customer choice box, the dashboard reflects the shares information of that customer.
- The piechart also gets updated accordingly.



Company	Qty	Price	Date
SUNWW Company (SUNWW)	40	$50.00	Nov 24, 2011
SUNWW Company (SUNWW)	50	$25.00	Jul 24, 2008
BW Corp (BWINC)	40	$20.00	Nov 21, 2010

Assumptions

You have completed the previous practice successfully.

Tasks

1. In the Projects window, expand the `BrokerTool_2Tier` project and verify the following.
   a. Under Source Packages, expand META-INF and verify that the `persistence.xml` file exists. Note the name of the persistent unit.
   b. Expand the `brokermodel` package and verify that the `Broker`, `Shares`, `Customer`, and `Stock` classes exist.
2. Create the `BrokerEntityManager` class and the DAO classes
   a. Under Source Packages, create a new package and name it `BrokerToolDAO`.

b. Create the `BrokerEntityManager` class. In this class, you will create an `EntityManager` object using the `EntityManagerFactory` class. Your class should have the following lines of code:

```
package brokerToolDAO;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceContext;

public class BrokerEntityManager {
     @PersistenceContext(unitName = "BrokerTool_2TierPU")
    private EntityManagerFactory emf;
    private EntityManager em;
   public BrokerEntityManager(){

}
    public void create(){


        emf = 
Persistence.createEntityManagerFactory("BrokerTool_2TierPU");
         em = emf.createEntityManager();
    }
    public void close(){
         em.close();
        emf.close();
    }
public EntityManager getEntityManager(){
    return em;
}
}
```

c. For the DAO class, first create an abstract class and name it `AbstractBrokerDAO`.

d. Create `BrokerDAO`, `CustomerDAO`, `StockDAO`, and `SharesDAO` classes that extend the `AbstractBrokerDAO` class.

e. Refer to the classes located in the `D:\labs\10-ApplyJPA\practices\practice02\brokertoolDAO` folder and code the classes that you have created in step d.

f. At the end of this task, your DAO classes and entity classes are ready to be used by the JavaFX classes.

3. In this and following tasks, you will modify the front end of the JavaFX classes. Complete the following steps to examine the `BrokerDashboard.fxml`.

   a. Expand the `brokertoolclient` package.

   b. Click `BrokerDashboard.fxml` to open it in the editor window.

   c. Identify the controller for the file.
   d. Identify the event handler for the customer choice box.
   e. Identify the IDs for the piechart and table.
4. Complete the following steps to modify `BrokerDashboard.java`, which is the controller of `BrokerDashboard.fxml`.
   a. Verify that the choice box, tableview, and piechart JavaFX UI elements are declared.
   b. Declare the `SharesDAO`, `CustomerDAO`, and `Customer` instances and make the required imports.

```
private SharesDAO sharefacade;
private CustomerDAO customerfacade;
private Customer cust;
```

   c. Modify the `brokerdashboard` constructor to initialize the `CustomerDAO` instance.

```
customerfacade=new CustomerDAO();
```

   d. Create an event handler for the customer choice box such that the event listener method fetches sales data for the selected customer. Refer to the following code:

```
customerChoiceBox.getSelectionModel().selectedIndexProperty().ad
dListener(new ChangeListener<Number>(){
            @Override
       public void changed(ObservableValue <? extends Number>
ov, Number value, Number newValue){
            if (newValue.intValue()>0 ){

sharesService.setUserAndPw(BrokerToolClientApp.getInstance().get
User(),BrokerToolClientApp.getInstance().getPassword(),
customerChoiceBox.getSelectionModel().getSelectedIndex()+1);
                if (sharesService.isRunning())
           sharesService.createTask();
             else
             sharesService.restart();
}
        }


  });
```

   e. Review the `GetSharesService.java` and `GetSharesTask.java` classes to figure out how the customer's shares data is retrieved from the database.
   f. Examine the `call` method of `GetSharesTask.java` that is equivalent to the run method of a Java thread.
5. Save the changes in the `BrokerDashboard.java` file, and build and run the project. Ensure that the JavaDB server is started.
   a. Verify the broker dashboard for sales data of customer 1.
   b. Select the third customer from the customer choice box and verify that the records of that customer are displayed in the broker dashboard.
   c. Verify that the piechart also changes correspondingly.

Practices for Lesson 11: Implementing a Multi-tier Design with RESTful Web Services

Chapter 11

Practices for Lesson 11

Practices Overview

In these practices, you will review how RESTful web services are implemented in a three-tier application. You will:

- Differentiate between two-tier and three-tier design
- Identify the features of web services
- Examine the BrokerToolServer application and review the implementation of its web services

Practice 11-1: Reviewing Basic Concepts of Java Web Services

Overview

In this practice, you will check your understanding about:

- Two-tier and three-tier architecture
- Java web services
- RESTful web services

Tasks

1. State whether the following statements about two-tier and three-tier design are true or false.
    a. In a two-tier design, the presentation logic, business logic, and data resources could be placed in a single system.
    b. One of the advantages of two tier design is that any changes to the business strategy result in changes to the business logic, which requires redeployment of all clients. This can be very costly and time-consuming.
    c. In a three-tier design, it is possible to change business logic without redeploying client software.
    d. An application server is a mandatory component for two-tier architecture.
    e. The features of load balancing and scaling can be applied easily to three-tier architecture.
2. Fill in the blanks in the following statements about web services.
    a. Web services are interoperable and extensible because of using _________.
    b. The two types of web services are SOAP-based web services and ____________ web services.
    c. In RESTful web services, resources are manipulated by a fixed set of four create, read, update, and delete operations that are represented by ___________________HTTP methods.
    d. REStful web services are based on the JSR-311 specification or the JAX-RS API and ______is a reference implementation of JAX-RS.
    e. A RESTful design could be appropriate when the web services are completely _____

3. Examine the following code snippet:

```
@Path("/hello")
public class Hello {
@GET
@Produces("text/plain")
public String sayHello() {
// Return some textual content
return "Hello World";
}
@POST
public void storeMessage(String message) {
      save(message)
}
}
```

a. Identify the annotation that denotes the URI for the Java class.

b. Identify the method in the snippet that will process HTTP GET requests.

c. Identify the annotation that is used to denote that a method will produce content identified by the "text/plain" MIME Media type.

Practice 11-2: (Summary Level) Examining `BrokerToolServer`'s Web Services

Overview

In this practice you examine the BrokerToolServer application and review the implementation of its web services. The objective is to identify how the BrokerTool application implements a three-tier design using web services.

Assumptions

NetBeans is running.

Tasks

1. In NetBeans, open all the projects required for this practice.
   a. Close all `BrokerTool` (`Model`, `Client`, or `Server`) projects that are open in NetBeans.
   b. Open the `BrokerToolModel`, `BrokerToolServer`, and `BrokerToolClient` projects located in the `D:\labs\11-MultiTier\practices\practice02` folder.
   c. Ensure that the `BrokerToolModel` project to the library of the `BrokerToolServer` project.
2. Identify and examine the entity classes of `BrokerToolModel` project.
3. Examine the RESTful web services in the `BrokerToolServer` project. Identify and explore all the files and classes that are related to a RESTful web service creation including the persistence unit and JPA related code.
4. To test the web services, deploy the `BrokerToolServer` project.
   a. Test the four RESTful web services: `BrokerFacadeREST`, `CustomerFacadeREST`, `SharesFacadeREST`, and `StockFacadeREST`.
   b. Observe the output for each web service in the web browser window.

Practice 11-2: (Detail Level) Examining `BrokerToolServer`'s Web Services

Overview

In this practice, you examine the BrokerToolServer application and review the implementation of its web services. The objective is to identify how the BrokerTool application implements a three-tier design using web services.

Assumptions

NetBeans is running.

Tasks

1. In NetBeans, open all the projects required for this practice.
   a. Close all `BrokerTool` (`Model`, `Client`, or `Server`) projects that are open in NetBeans.
   b. Open the `BrokerToolModel`, `BrokerToolServer`, and `BrokerToolClient` projects located in the `D:\labs\11-MultiTier\practices\practice02` folder.
   c. Ensure that the `BrokerToolModel` project is added to the library of `BrokerToolServer` project.
2. Examine the `BrokerToolModel` project.
   a. In the Projects window, expand `BrokerToolModel > Source Packages > com.brokertool.model`.
   b. Examine the four classes: `Broker`, `Customer`, `Shares`, and `Stock`.
   c. Verify that these are the Entity classes that have been created from the BrokerTool database.
3. Examine the RESTful web services in the `BrokerToolServer` project. The `BrokerToolServer` project is a web application.
   a. Expand the `BrokerToolServer > Configuration Files` node in the Projects window and select `persistence.xml`.
   b. Examine the persistence unit configured in the `persistence.xml` file.
   c. In the `persistence.xml` file, check that the entity classes are added and that the check box to use JTA is selected.
   d. Expand the `BrokerToolServer > Source Packages > com.brokertool.service` node in the Projects window.
   e. Examine the `AbstractFacade.java` file. Identify JPA-related code in all the methods defined in the class.

f. Review the four RESTful web services: `BrokerFacadeREST`, `CustomerFacadeREST`, `SharesFacadeREST`, and `StockFacadeREST` in terms of their:

   1) `@Path` annotation

   2) HTTP operations and their corresponding methods

   3) `@Produces` annotation

4. Test the RESTful web services.

   a. Ensure that the Java DB server has been started and that the GlassFish server is running.

   b. Build the `BrokerToolServer` project.

   c. Deploy `BrokerToolServer`.

   d. In the Projects window, expand `RESTful Web Services.`

   e. Right-click `BrokerFacadeREST` and select Test Resource URI and view the output in the browser window. Observe the URL of the web service in the browser address bar.

   f. Repeat step e for the other RESTful web services: `CustomerFacadeREST`, `SharesFacadeREST`, and `StockFacadeREST`.

   g. Close the `BrokerToolServer` project in NetBeans.

Solutions For Practice 1

Solutions for Tasks

1. Correct answers are listed below:
   a. True
   b. False
   c. True
   d. False
   e. True
2. The correct answers are listed below:
   a. XML
   b. RESTful
   c. PUT, GET, POST, and DELETE
   d. Jersey
   e. Stateless
3. The correct answers are listed below:
   a. @Path
   b. public String sayHello()
   c. @Produces

Practices for Lesson 12: Connecting to a RESTful Web Service

Chapter 12

Practices for Lesson 12

Practices Overview

In these practices, you will:

- Test the RESTful web services of BrokerTool and examine their output in JSON and XML format
- Create a RESTful web service client in the BrokerTool client application

Practice 12-1: (Summary Level) Testing RESTful Web Services

Overview

In this practice, you will test the RESTful Web Services created in BrokerTool Application.

Assumptions

NetBeans is running.

Tasks

1. Start NetBeans and browse to the `D:\labs\12-connectRest\practices\practice_01` folder. Open the `BrokerToolServer Project NetBeans`. Ensure that `BrokerToolModel` project also opens in NetBeans.
2. In the Projects window, expand BrokerToolServer > RESTful Web Services and view the list of RESTful web services existing in the project.
3. To test the web services, you must generate the web services test client. Perform the following steps:
   a. Select the `BrokerToolServer` project.
   b. Right-click and select Test RESTful WebServices. In the Configure REST test Client dialog box, select "Web Test Client in project" and click Browse.
   c. Select `BrokerToolServer`, in the Select Project dialog box, and click OK.
   d. In the Configure Rest Web Client window, click OK. The server starts and the application is deployed.
4. To test the Web Services Client, perform the following steps:
   a. When deployment is complete, the browser displays your application, with a link for each of the web services.
   b. In the left side of the window, select one resource node, such as `com.brokertool.model.customer`.
   c. In the "Choose method to test" field, select GET.
   d. Click Test. The test client sends a request and displays the result in the Test Output section.
5. Close the `BrokerToolServer` and `BrokerToolModel` projects in NetBeans.

Practice 12-1: (Detail Level) Testing RESTful Web Services

Overview

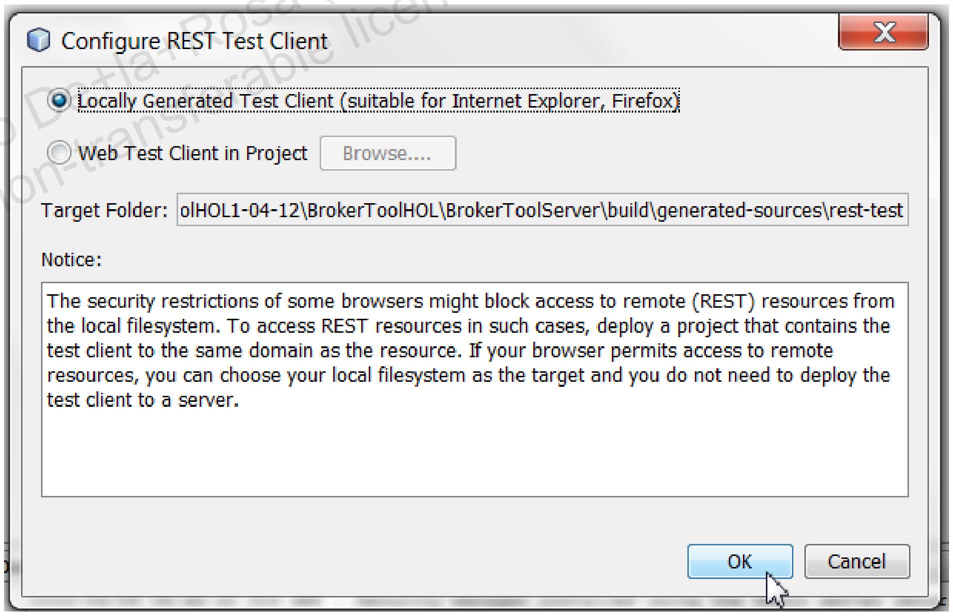
In this practice, you will test the RESTful Web Services created in BrokerTool Application.

Assumptions

NetBeans is running.

Tasks

1. In NetBeans, complete the following steps to open the `BrokerTool` application:
   a. Close all BrokerTool (Client, Model, and Server) projects that are open in NetBeans.
   b. Select File > Open Project.
   c. In the “Open Project” dialog box, browse to the `D:\labs\12-ConnectRest\practices\practice_01` folder.
   d. Select and open the `BrokerToolServer and BrokerModel` project.
2. In the Projects window, expand `BrokerToolServer > RESTful Web Services` and view the list of RESTful web services existing in the project.
3. To generate the web services test client, complete the following steps:
   a. Select the `BrokerToolServer` project.
   b. Right-click and select Test RESTful WebServices. The Configure REST test Client dialog box opens, as shown in the following figure.



Configure REST Test Client

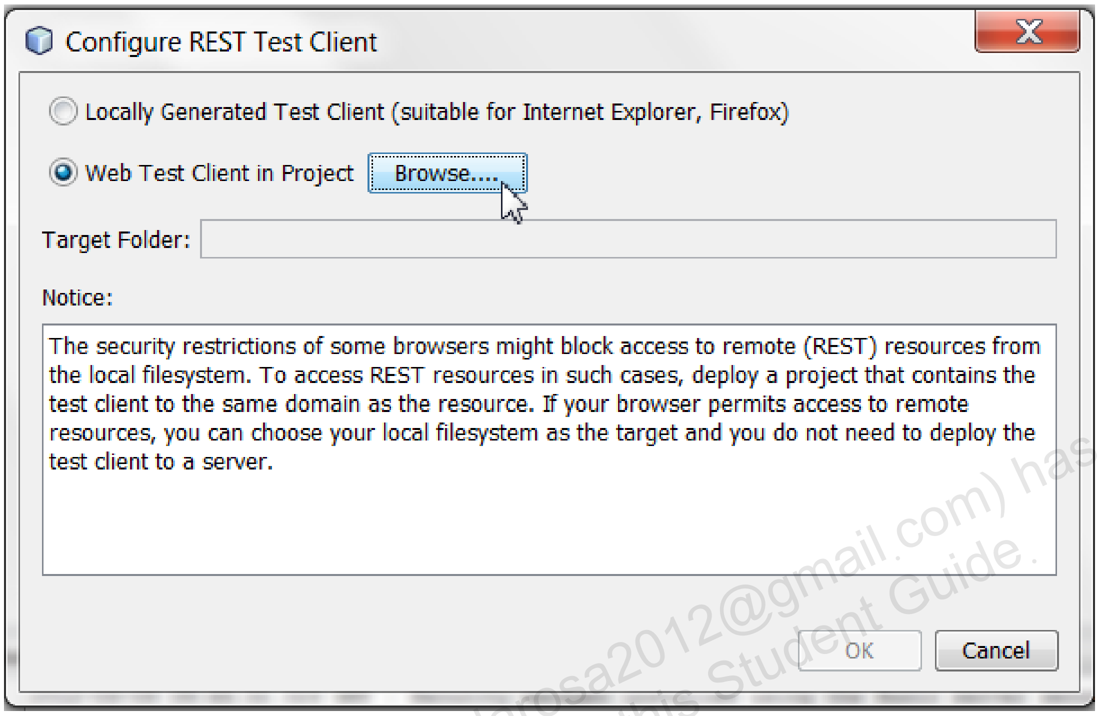
(● Locally Generated Test Client (suitable for Internet Explorer, Firefox))
○ Web Test Client in Project Browse....

Target Folder: olHOL1-04-12\BrokerToolHOL\BrokerToolServer\build\generated-sources\rest-test

Notice:

The security restrictions of some browsers might block access to remote (REST) resources from the local filesystem. To access REST resources in such cases, deploy a project that contains the test client to the same domain as the resource. If your browser permits access to remote resources, you can choose your local filesystem as the target and you do not need to deploy the test client to a server.

c. Select “Web Test Client in Project” and click Browse.



Configure REST Test Client

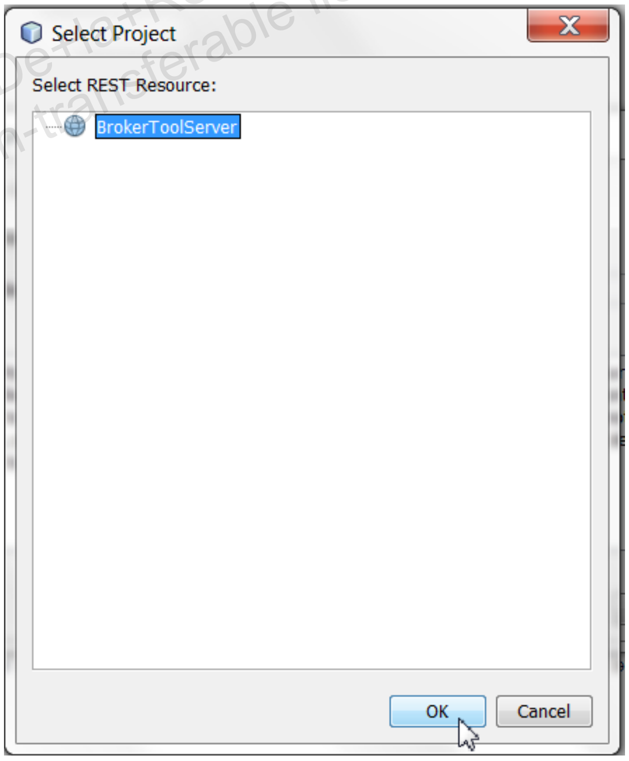
Locally Generated Test Client (suitable for Internet Explorer, Firefox) [✓ Web Test Client in Project]

Target Folder:

Notice:

The security restrictions of some browsers might block access to remote (REST) resources from the local filesystem. To access REST resources in such cases, deploy a project that contains the test client to the same domain as the resource. If your browser permits access to remote resources, you can choose your local filesystem as the target and you do not need to deploy the test client to a server.

d. Select `BrokerToolServer` in the Select Project dialog box and click OK.

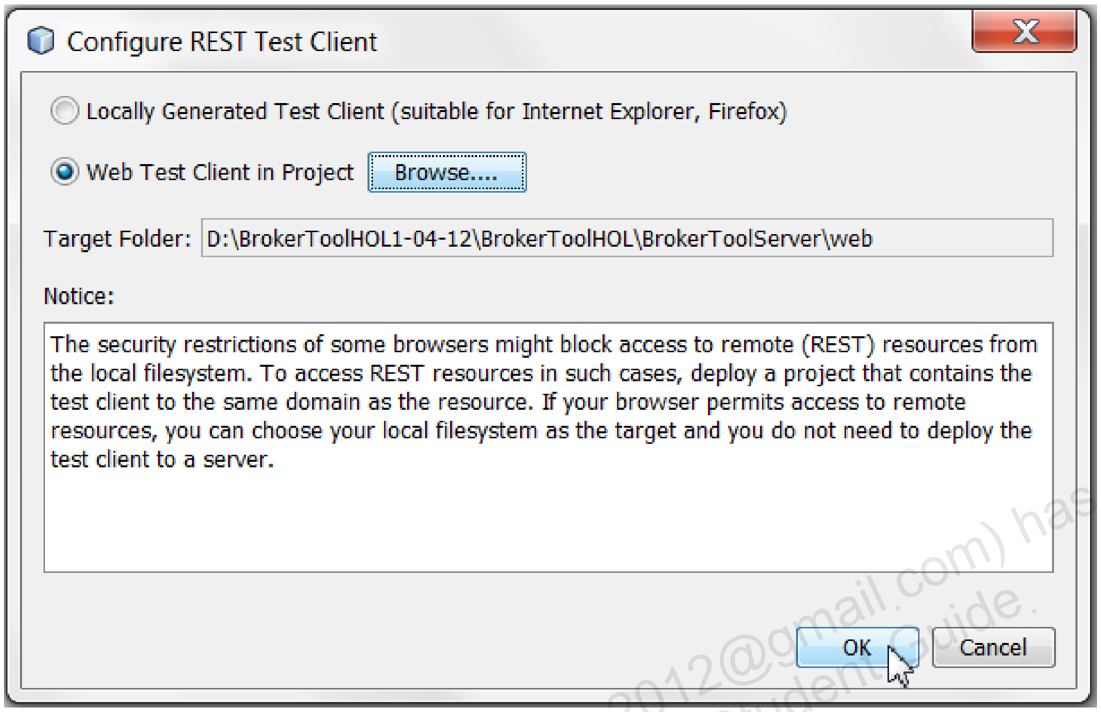


Select Project

Select REST Resource:

BrokerToolServer

e. In the Configure Rest Web Client dialog box, click OK.



Locally Generated Test Client (suitable for Internet Explorer, Firefox)

[✓ Web Test Client in Project] Browse....

Target Folder: D:\BrokerToolHOL1-04-12\BrokerToolHOL\BrokerToolServer\web

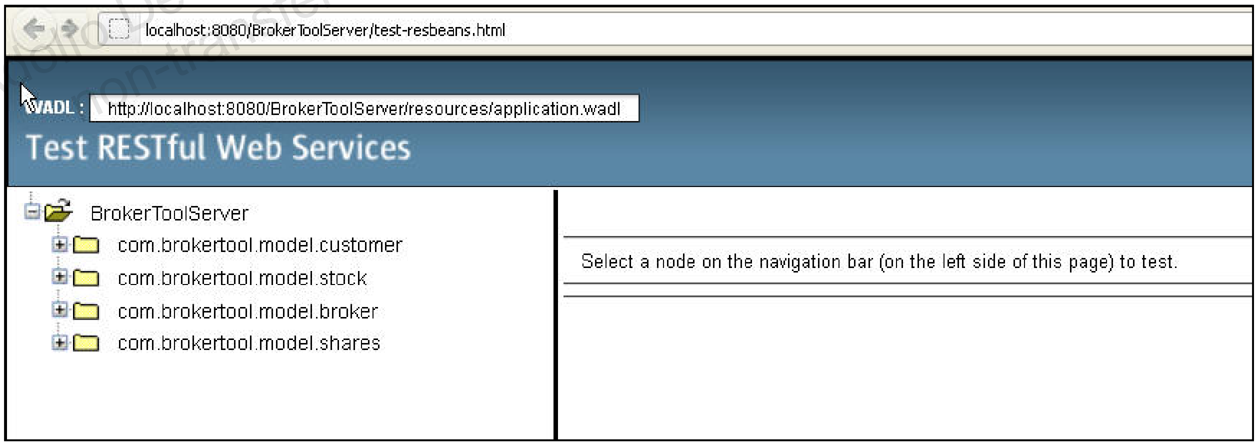
Notice:

The security restrictions of some browsers might block access to remote (REST) resources from the local filesystem. To access REST resources in such cases, deploy a project that contains the test client to the same domain as the resource. If your browser permits access to remote resources, you can choose your local filesystem as the target and you do not need to deploy the test client to a server.

The server starts and the application is deployed.

4. To test the Web Services Client, perform the following steps:

a. When deployment is complete, the browser displays your application, with a link for each of the web services.



localhost:8080/BrokerToolServer/test-resbeans.html

WADL : http://localhost:8080/BrokerToolServer/resources/application.wadl

Test RESTful Web Services

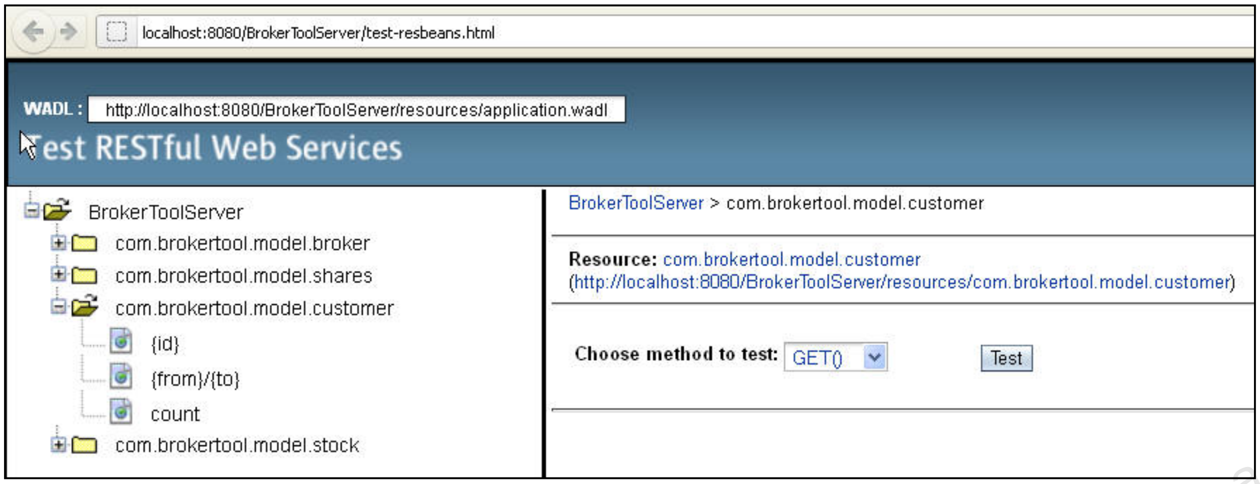
- BrokerToolServer
  - com.brokertool.model.customer
  - com.brokertool.model.stock
  - com.brokertool.model.broker
  - com.brokertool.model.shares

Select a node on the navigation bar (on the left side of this page) to test.

b. In the left side of the window, select one resource node, such as `com.brokertool.model.customer`

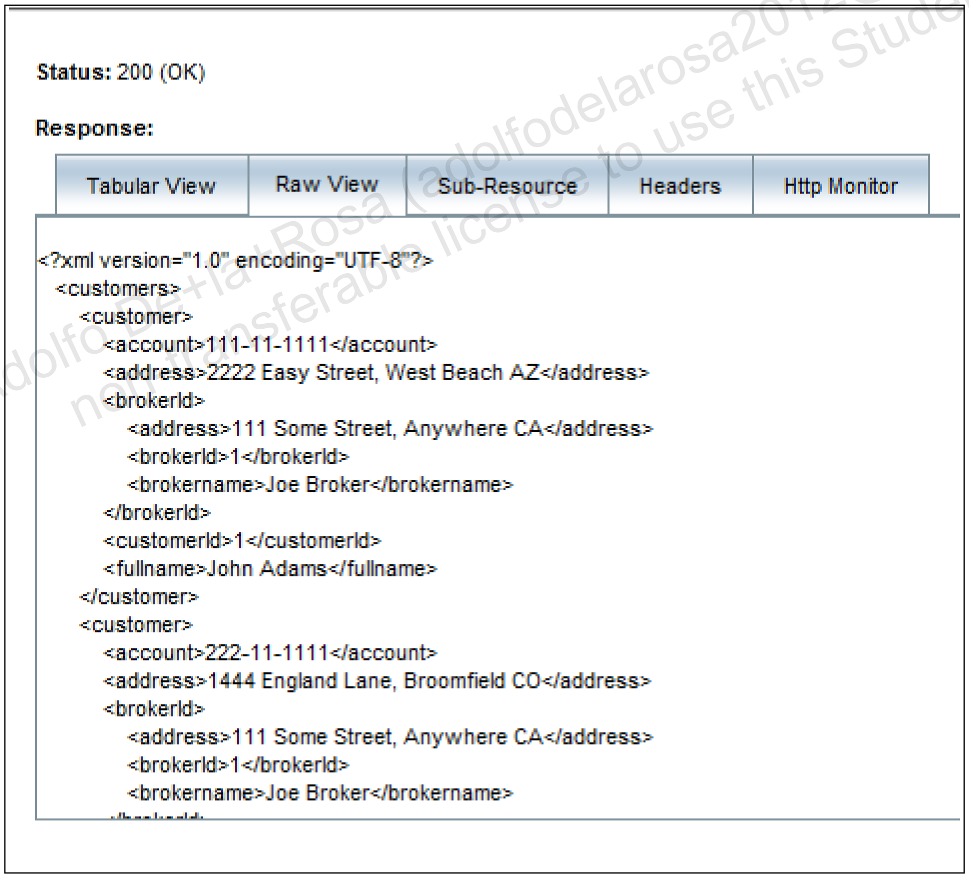
c. In the "Choose method to test" field, select GET.

d. Click Test.


localhost:8080/BrokerToolServer/test-resbeans.html
WADL : http://localhost:8080/BrokerToolServer/resources/application.wadl
Test RESTful Web Services
BrokerToolServer
com.brokertool.model.broker
com.brokertool.model.shares
com.brokertool.model.customer
{id}
{from}/{to}
count
com.brokertool.model.stock
BrokerToolServer > com.brokertool.model.customer
Resource: com.brokertool.model.customer
(http://localhost:8080/BrokerToolServer/resources/com.brokertool.model.customer)
Choose method to test: GET()
Test

The test client sends a request and displays the result in the Test Output section.

5. Examine the output, the response to an application/xml request. The test client displays the Raw View by default as shown in the following image.



Status: 200 (OK)

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
  <customers>
    <customer>
      <account>111-11-1111</account>
      <address>2222 Easy Street, West Beach AZ</address>
      <brokerId>
        <address>111 Some Street, Anywhere CA</address>
        <brokerId>1</brokerId>
        <brokername>Joe Broker</brokername>
      </brokerId>
      <customerId>1</customerId>
      <fullname>John Adams</fullname>
    </customer>
    <customer>
      <account>222-11-1111</account>
      <address>1444 England Lane, Broomfield CO</address>
      <brokerId>
        <address>111 Some Street, Anywhere CA</address>
        <brokerId>1</brokerId>
        <brokername>Joe Broker</brokername>
```

6. Close the `BrokerToolServer` and `BrokerToolModel` projects in NetBeans.

Practice 12-2: (Summary Level) Creating a RESTful Web Services Client

Overview

In this practice, you will create a RESTful web service client to consume the `com.brokertool.model.shares` web service of the `BrokerToolServer` program.

Assumptions

NetBeans is running.

Tasks

1. Close all BrokerTool (Client, Model, or Server) projects that are open in NetBeans.
2. Open `BrokerToolServer`, `BrokerToolModel`, and `BrokerToolClient3` located in the `D:\labs\12-ConnectRest\practices\practice_02` folder. The project will show some compilation errors, which you can ignore at this point.
3. Deploy the `BrokerToolServer` project on the GlassFish server. The RESTful web services of `BrokerToolServer` are also published.
4. Right-click `BrokerToolClient3` and select New > Other > Web service > RESTful Java Client, and click Next. Perform the following in the New Restful Java Client dialog box:
   a. For Class Name, use `GetSharesJerseyClient`.
   b. For Package, use `brokertoolclient`.
   c. For the REST resource, select From Project.
   d. Click Browse and select `BrokerToolServer`.
   e. Expand the BrokerToolServer application and select `SharesFacadeREST[com.brokertool.model.shares]`.
   f. Click OK.
   g. Click Finish.

   `GetSharesJerseyClient.java` opens in the code editor.
5. Examine `GetSharesJerseyClient.java`. Identify all the methods of the web service that map to a GET, a POST, a PUT, or a DELETE operation.
6. You will place the web service client code in a JavaFX file. Perform the following steps:
   a. Open `GetSharesTask.java` and locate the `call` method.
   b. Add the following lines of code in the `call` method to consume the Shares web service:

```
WebResource webResource =
client.resource("http://localhost:8080/BrokerToolServer/resource
s").path("com.brokertool.model.shares");
        ClientResponse response =
webResource.get(ClientResponse.class);
```

c. The remaining code in the call method is for returning the correct data that is obtained from the `findAll` method or `get` operation.

d. Save `GetSharesTask.java.`

7. Build and run the `BrokerToolClient3` project.
8. Examine the BrokerDashboard, displaying the Shares data in a table and a piechart for the first customer in the Customers choice box.
9. In this task, you will modify `OrderForm.java` such that when a user submits an order through a JavaFX form, a new record is inserted in the shares table via a POST operation in the Shares RESTful web service.

a. In `OrderForm.java`, locate the `saveOrderTask Task` meant for invoking the RESTful web service.

b. Add the following lines of code in the `call` method of the task, to create an order by using the Shares web service:

```
@Override
 protected Object call() throws Exception {
 Client client = Client.create(CONFIG);
 client.addFilter(new
HTTPBasicAuthFilter(BrokerToolClientApp.getInstance().getUser(),

BrokerToolClientApp.getInstance().getPassword()));
                WebResource orderWebResource =
client.resource("http://localhost:8080/BrokerToolServer/resource
s").path("com.brokertool.model.shares");

orderWebResource.type(javax.ws.rs.core.MediaType.APPLICATION_JSO
N).post(shares);
                return null;
            }
```

10. Run the `BrokerToolClient3` project.
11. Create a new order for a customer and verify that the order data is visible in the BrokerDashboard table for the particular customer.
12. Close all the three projects in NetBeans.

Optional: Identify two other instances in `OrderForm.java` where Customer and Stocks RESTful web services are consumed.

Practice 12-2: (Detail Level) Creating a RESTful Web Services Client

Overview

In this practice, you will create a RESTful web service client to consume the `com.brokertool.model.shares` web service of the `BrokerToolServer` program.

Assumptions

NetBeans is running.

Tasks

1. Close all BrokerTool (Client, Model, or Server) projects that are open in NetBeans.
2. In the Projects window, select File >Open and open the following projects: `BrokerToolServer`, `BrokerToolModel`, and `BrokerToolClient3` located in the `D:\labs\12-ConnectRest\practices\practice_02` folder. The project will show some compilation errors, which you can ignore at this point.
3. Deploy the `BrokerToolServer` project on the GlassFish server. The RESTful web services of `BrokerToolServer` are also published.
4. Complete the following steps to create RESTful client in the `BrokerToolClient3` project
   a. Right-click `BrokerToolClient3` and select New > Other > Web service > RESTful Java Client.
   b. Click Next.
   c. Do the following in the New Restful Java Client dialog box:
      1) For Class Name, select `GetSharesJerseyClient`.
      2) For Package, use `brokertoolclient`.
      3) Select the REST resource: From Project.
      4) Click Browse and select `BrokerToolServer`.
      5) Expand the BrokerToolServer application and select `SharesFacadeREST[com.brokertool.model.shares]`.
      6) Click OK.
      7) Click Finish.

   `GetSharesJerseyClient.java` opens in the code editor.
5. Examine `GetSharesJerseyClient.java.`
   a. Identify the code snippet where the WebResource is initialized with the URI and path of the RESTful service.

   Observe that all the methods of the web service that map to a GET, a POST, a PUT, or a DELETE operation, are present. Each of the web methods has an `application/XML` and `application/JSON` format. Because in `BrokerToolClient3` the user interface is in JavaFX, the values returned by the web methods have to be displayed in a JavaFX table.

   b. Examine the `findAll` method.

c. Close the `GetSharesJerseyClient.java` file.

6. You will place the web service client code in a JavaFX file. Complete the following steps:

   a. Open `GetSharesTask.java` and locate the `call` method.

   b. Add the following lines of code in the `call` method to consume the Shares web service:

```
WebResource webResource =
client.resource("http://localhost:8080/BrokerToolServer/resource
s").path("com.brokertool.model.shares");
        ClientResponse response =
webResource.get(ClientResponse.class);
```

   c. The remaining code in the call method is for returning the correct data that is obtained from the `findAll` method or `get` operation.

   d. Save `GetSharesTask.java.`

7. Build and run the `BrokerToolClient3` project.

8. Examine the BrokerDashboard, displaying the Shares data in a table and a piechart for the first customer in the Customers choice box.

9. In this task, you will modify `OrderForm.java` such that when a user submits an order through a JavaFX form, a new record in inserted in the shares table via a POST operation in the Shares RESTful web service

   a. In `OrderForm.java`, locate the `saveOrderTask Task` meant for invoking the RESTful web service.

   b. Add the following lines of code in the `call` method of the task, to create an order by using the Shares web service:

```
@Override
 protected Object call() throws Exception {
 Client client = Client.create(CONFIG);
 client.addFilter(new
HTTPBasicAuthFilter(BrokerToolClientApp.getInstance().getUser(),

BrokerToolClientApp.getInstance().getPassword()));
                WebResource orderWebResource =
client.resource("http://localhost:8080/BrokerToolServer/resource
s").path("com.brokertool.model.shares");

orderWebResource.type(javax.ws.rs.core.MediaType.APPLICATION_JSO
N).post(shares);
                return null;
            }
```

   c. Observe that, for creating a new order, the POST operation is invoked.

10. Run the `BrokerToolClient3` project.

11. Create a new order for a customer and verify that the order data is visible in the BrokerDashboard table for the particular customer.
12. Close all the three projects in NetBeans.

    Optional: Identify two other instances in `OrderForm.java` where Customer and Stocks RESTful web services are consumed.

Practices for Lesson 13: Packaging and Deploying Applications

Chapter 13

Practices for Lesson 13

Practices Overview

In this practice, you will use NetBeans and a Java EE Web Application to deploy the `TextViewer` application on an application server. Then, you will access the TextViewer application in a web browser as an embedded application or via Java Web Start.

Practice 13-1: (Summary Level) Deploying an Application in a Web App

Overview

In this practice, you will take a JavaFX application and deploy it on a Java EE server.

Assumptions

You have listened to the lecture portion of this lesson.

Summary

You will take the `TextViewer` NetBeans project and build an application. Using the files created by NetBeans, load the application into a Java EE web project, deploy the application, and run it from an application server.

Note: The `TextViewer` application has been self-signed so that it can access the local file system. Detailed information on signing application is covered in the lesson titled "*Securing an Application*".

Tasks

1. Open the `TextViewer` project in the `resources` directory in the `labs` directory for this lesson.
2. Clean and build the project. This will create a `dist` directory for the project. This directory will create a `TextViewer.jar`, `TextViewer.html`, `TextViewer.jnlp`, and related files and directories required to deploy a JavaFX application in a web browser.
3. Open the `DeployApp` project in the `practice01` directory.
4. Copy the contents of the `resources\TextViewer\dist` to the `practice01\DeployApp\web\deploy-all` directory.
5. Edit the `index.jsp` file to add a link to `TextViewer.html` file.
6. Build the `DeployApp` project.
7. Deploy and run the `DeployApp` project. This should launch a web browser.
8. From your web browser, launch and test the `TextViewer` application both as an embedded application and as a Java Web Start application. Because the application is signed, it can access local resources.
9. Close the `DeployApp` project.

Practice 13-1: (Detail Level) Deploying an Application in a Web App

Overview

In this practice, you will take a JavaFX application and deploy it on a Java EE server.

Assumptions

You have listened to the lecture portion of this lesson.

Summary

You will take the `TextViewer` NetBeans project and build an application. Using the files created by NetBeans, load the application into a Java EE Web project, deploy the application, and run it from an application server.

Note: The `TextViewer` application has been self-signed so that it can access the local file system. Detailed information on signing application is covered in the lesson titled "*Securing an Application*".

Tasks

1. Open the `TextViewer` project in the `resources` directory in the `labs` directory for this lesson.
2. Clean and build the project. This will create a `dist` directory for the project. This directory will create a `TextViewer.jar`, `TextViewer.html`, `TextViewer.jnlp`, and related files and directories required to deploy a JavaFX application in a web browser.
3. Open the `DeployApp` project in the `practice01` directory. Review the contents of the project.
4. Open the `Web Pages` folder.
5. Open the `index.jsp` file for editing.

   Note: There is a `deploy-all` directory under `Web Pages`. This is where we will deploy the `TextViewer` application.
6. Copy the contents of the `resources\TextViewer\dist` to the `practice01\DeployApp\web\deploy-all` directory.
7. Edit the `index.jsp` file to add a link to `TextViewer.html` file. For example:

   ```
   <a href="deploy-all\TextViewer.html">TextViewer All</a>
   ```
8. Build the `DeployApp` project.
9. Deploy and run the `DeployApp` project. This should launch a web browser.
10. From your web browser, launch and test the `TextViewer` application both as an embedded application and as a Java Web Start application. Because the application is signed, it can access local resources.
11. Close the `DeployApp` project.

Practice 13-2: (Summary Level) Deploying an Embedded Application Only (Browser Only)

Overview

Deploy the `TextViewer` as an embedded application.

Assumptions

You have listened to the lecture portion of this lesson and completed the previous practice.

Summary

Perform the same tasks as in the previous practice. However, in this practice deploy the application as an embedded application only.

Tasks

1. Open the `TextViewer` project in the `resources` directory in the `labs` directory for this lesson.
2. Clean and build the project. This will create a `dist` directory for the project. This directory will create a `TextViewer.jar`, `TextViewer.html`, `TextViewer.jnlp`, and related files and directories required to deploy a JavaFX application in a web browser.
3. Open the `DeployApp` project in the `practice02` directory.
4. Copy the contents of the `resources\TextViewer\dist` to the `practice02\DeployApp\web\deploy-embedded` directory.
5. Edit the `index.jsp` file to add a link to `TextViewer.html` file. For example:
```
    <a href="deploy-embedded\TextViewer.html">TextViewer All</a>
```
6. Edit the `deploy-embedded\TextViewer.html` file.
7. Remove any `<script>` tags or other links related to Java Web Start.
8. Save the file.
9. Build the `DeployApp` project.
10. Deploy and run the `DeployApp` project. This should launch a web browser.
11. From your web browser, launch and test the `TextViewer` application as an embedded application. Because the application is signed, it can access local resources.
12. Close the `DeployApp` project.

Practice 13-2: (Detail Level) Deploying an Embedded Application Only (Browser Only)

Overview

Deploy the `TextViewer` as an embedded application.

Assumptions

You have listened to the lecture portion of this lesson and completed the previous practice.

Summary

Perform the same tasks as in the previous practice. However, in this practice, deploy the application as an embedded application only.

Tasks

1. Open the `TextViewer` project in the `resources` directory in the `labs` directory for this lesson.
2. Clean and build the project. This will create a `dist` directory for the project. This directory will create a `TextViewer.jar`, `TextViewer.html`, `TextViewer.jnlp`, and related files and directories required to deploy a JavaFX application in a web browser.
3. Open the `DeployApp` project in the `practice02` directory. Review the contents of the project.
4. Open the `Web Pages` folder.
5. Open the `index.jsp` file for editing.
6. Note that there is a `deploy-embedded` directory under `Web Pages`. This is where you will deploy the `TextViewer` application.
7. Copy the contents of the `resources\TextViewer\dist` to the `practice02\DeployApp\web\deploy-embedded` directory.
8. Edit the `index.jsp` file to add a link to `TextViewer.html` file. For example:

```
<a href="deploy-embedded\TextViewer.html">TextViewer All</a>
```

9. Edit the `deploy-embedded\TextViewer.html` file.
10. Remove any `<script>` tags or other links related to Java Web Start.

```
<script>
    function launchApplication(jnlpfile) {
        dtjava.launch(            {
                url : 'TextViewer.jnlp',
                jnlp_content :
'PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0idXRmLTgiPz4NCjxqbmxwIHN
wZWM9IjEuMCIgeG1sbnM6amZ4PSJodHRwOi8vamF2YWZ4LmNvbSIgaHJlZj0iVGV
4dFZpZXdlci5qbmxwIj4NCiAgPGluZm9ybWF0aW9uPg0KICAgIDx0aXRsZT5UZXh
0Vmlld2VyPC90aXRsZT4NCiAgICA8dmVuZG9yPm13MTE5MjU8L3ZlbmRvcj4NCiA
gICA8ZGVzY3JpcHRpb24+U2FtcGxlIEphdmFGWCAyLjAgYXBwbGljYXRpb24uPC9
kZXNjcmlwdGlvbj4NCiAgICA8b2ZmbGluZS1hbGxvd2VkLz4NCiAgPC9pbmZvcm1
hdGlvbj4NCiAgPHJlc291cmNlcz4NCiAgICA8amZ4OmphdmFmeC1ydW50aW1lIHZ
lcnNpb249IjIuMCsiIGhyZWY9Imh0dHA6Ly9qYXZhZGwuc3VuLmNvbS93ZWJhcHB
zL2Rvd25sb2FkL0dldEZpbGUvamF2YWZ4LWxhdGVzdC93aW5kb3dzLWk1ODYvamF
2YWZ4Mi5qbmxwIi8+DQogIDwvcmVzb3VyY2VzPg0KICA8cmVzb3VyY2VzPg0KICA
gIDxqMnNlIHZlcnNpb249IjEuNisiIGhyZWY9Imh0dHA6Ly9qYXZhLnN1bi5jb20
vcHJvZHVjdHMvYXV0b2RsL2oyc2UiLz4NCiAgICA8amFyIGhyZWY9IlRleHRWaWV
3ZXIuamFyIiBzaXplPSIyMTg0MCIgZG93bmxvYWQ9ImVhZ2VyIiAvPg0KICA8L3J
lc291cmNlcz4NCjxzZWN1cml0eT4NCiAgPGFsbC1wZXJtaXNzaW9ucy8+DQo8L3N
lY3VyaXR5Pg0KICA8YXBwbGV0LWRlc2MgIHdpZHRoPSI4MDAiIGhlaWdodD0iNjA
wIiBtYWluLWNsYXNzPSJjb20uamF2YWZ4Lm1haW4uTm9KYXZhRlhGYWxsYmFjayI
gIG5hbWU9IlRleHRWaWV3ZXIiID4NCiAgICA8cGFyYW0gbmFtZT0icmVxdWlyZWR
GWFZlcnNpb24iIHZhbHVlPSIyLjArIi8+DQogIDwvYXBwbGV0Pg0KICA8amZ4Omp
hdmFmeC1kZXNjICB3aWR0aD0iODAwIiBoZWlnaHQ9IjYwMCIgbWFpbi1jbGFzcz0
iY29tLmV4YW1wbGUuZ3VpLlRleHRWaWV3ZXIiICBuYW1lPSJUZXh0Vmlld2VyIiA
vPg0KICA8dXBkYXRlIGNoZWNrPSJiYWNrZ3JvdW5kIi8+DQo8L2pubHA+DQo='
         },
         {
            javafx : '2.0+'
         },
         {}
      );
      return false;
   }
</script>
```

```
And the following link:
  <b>Webstart:</b> <a href='TextViewer.jnlp' onclick="return
launchApplication('TextViewer.jnlp');">click to launch this app
as webstart</a><br><hr><br>
```

11. Save the file.
12. Build the `DeployApp` project.
13. Deploy and run the `DeployApp` project. This should launch a web browser.
14. From your web browser, launch and test the `TextViewer` application as an embedded application. Because the application is signed, it can access local resources.
15. Close the `DeployApp` project.

Practice 13-3: (Summary Level) Deploying Java Web Start Only

Overview

Deploy the `TextViewer` as a Java Web Start application.

Assumptions

You have listened to the lecture portion of this lesson and completed the previous practice.

Summary

Perform the same tasks as in the previous practice. However, in this practice deploy the application as a Java Web Start application only.

Tasks

1. Open the `TextViewer` project in the `resources` directory in the `labs` directory for this lesson.
2. Clean and build the project. This will create a `dist` directory for the project. This directory will create a `TextViewer.jar`, `TextViewer.html`, `TextViewer.jnlp`, and related files and directories required to deploy a JavaFX application in a web browser.
3. Open the `DeployApp` project in the `practice03` directory.
4. Open the `Web Pages` folder.
5. Copy the contents of the `resources\TextViewer\dist` to the `practice03\DeployApp\web\deploy-webstart` directory.
6. Edit the `index.jsp` file to add a link to `TextViewer.html` file.
7. Edit the `deploy-webstart\TextViewer.html` file.
8. Remove any `<script>` tags or other elements related to running the application embedded.
9. Save the file.
10. Build the `DeployApp` project.
11. Deploy and run the `DeployApp` project. This should launch a web browser.
12. From your web browser, launch and test the `TextViewer` application as a Java Web Start application. Because the application is signed, it can access local resources.
13. Close the `DeployApp` project.

Practice 13-3: (Detail Level) Deploying Java Web Start Only

Overview

Deploy the `TextViewer` as a Java Web Start application.

Assumptions

You have listened to the lecture portion of this lesson and completed the previous practice.

Summary

Perform the same tasks as in the previous practice. However, in this practice deploy the application as a Java Web Start application only.

Tasks

1. Open the `TextViewer` project in the `resources` directory in the `labs` directory for this lesson.
2. Clean and build the project. This will create a `dist` directory for the project. This directory will create a `TextViewer.jar`, `TextViewer.html`, `TextViewer.jnlp`, and related files and directories required to deploy a JavaFX application in a web browser.
3. Open the `DeployApp` project in the `practice03` directory. Review the contents of the project.
4. Open the `Web Pages` folder.
5. Open the `index.jsp` file for editing.
6. Note that there is a `deploy-webstart` directory under `Web Pages`. This is where you will deploy the `TextViewer` application.
7. Copy the contents of the `resources\TextViewer\dist` to the `practice03\DeployApp\web\deploy-webstart` directory.
8. Edit the `index.jsp` file to add a link to `TextViewer.html` file. For example:

   ```
   <a href="deploy-webstart\TextViewer.html">TextViewer All</a>
   ```
9. Edit the `deploy-webstart\TextViewer.html` file.
10. Remove any `<script>` tags or other elements related to running the application embedded.

    ```
    <script>
        function javafxEmbed() {
            dtjava.embed(
                {
                    url : 'TextViewer.jnlp',
                    placeholder : 'javafx-app-placeholder',
                    width : 800,
                    height : 600,
                    jnlp_content :
    ```

```
'PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0idXRmLTgiPz4NCjxqbmxwIHN
wZWM9IjEuMCIgeG1sbnM6amZ4PSJodHRwOi8vamF2YWZ4LmNvbSIgaHJlZj0iVGV
4dFZpZXdlci5qbmxwIj4NCiAgPGluZm9ybWF0aW9uPg0KICAgIDx0aXRsZT5UZXh
0Vmlld2VyPC90aXRsZT4NCiAgICA8dmVuZG9yPm13MTE5MjU8L3ZlbmRvcj4NCiA
gICA8ZGVzY3JpcHRpb24+U2FtcGxlIEphdmFGWCAyLjAgYXBwbGljYXRpb24uPC9
kZXNjcmlwdGlvbj4NCiAgICA8b2ZmbGluZS1hbGxvd2VkLz4NCiAgPC9pbmZvcm1
hdGlvbj4NCiAgPHJlc291cmNlcz4NCiAgICA8amZ4OmphdmFmeC1ydW50aW1lIHZ
lcnNpb249IjIuMCsiIGhyZWY9Imh0dHA6Ly9qYXZhZGwuc3VuLmNvbS93ZWJhcHB
zL2Rvd25sb2FkL0dldEZpbGUvamF2YWZ4LWxhdGVzdC93aW5kb3dzLWk1ODYvamF
2YWZ4Mi5qbmxwIi8+DQogIDwvcmVzb3VyY2VzPg0KICA8cmVzb3VyY2VzPg0KICA
gIDxqMnNlIHZlcnNpb249IjEuNisiIGhyZWY9Imh0dHA6Ly9qYXZhLnN1bi5jb20
vcHJvZHVjdHMvYXV0b2RsL2oyc2UiLz4NCiAgICA8amFyIGhyZWY9IlRleHRWaWV
3ZXIuamFyIiBzaXplPSIyMTg0MCIgZG93bmxvYWQ9ImVhZ2VyIiAvPg0KICA8L3J
lc291cmNlcz4NCjxzZWN1cml0eT4NCiAgPGFsbC1wZXJtaXNzaW9ucy8+DQo8L3N
lY3VyaXR5Pg0KICA8YXBwbGV0LWRlc2MgIHdpZHRoPSI4MDAiIGhlaWdodD0iNjA
wIiBtYWluLWNsYXNzPSJjb20uamF2YWZ4Lm1haW4uTm9KYXZhRlhGYWxsYmFjayI
gIG5hbWU9IlRleHRWaWV3ZXIiID4NCiAgICA8cGFyYW0gbmFtZT0icmVxdWlyZWR
GWFZlcnNpb24iIHZhbHVlPSIyLjArIi8+DQogIDwvYXBwbGV0Pg0KICA8amZ4Omp
hdmFmeC1kZXNjICB3aWR0aD0iODAwIiBoZWlnaHQ9IjYwMCIgbWFpbi1jbGFzcz0
iY29tLmV4YW1wbGUuZ3VpLlRleHRWaWV3ZXIiICBuYW1lPSJUZXh0Vmlld2VyIiA
vPg0KICA8dXBkYXRlIGNoZWNrPSJiYWNrZ3JvdW5kIi8+DQo8L2pubHA+DQo='
             },
             {
                 javafx : '2.0+'
             },
             {}
         );
     }
     <!-- Embed FX application into web page once page is loaded
-->
     dtjava.addOnloadCallback(javafxEmbed);
</script>

And the div for the embedded application:
<!-- Applet will be inserted here -->
<div id='javafx-app-placeholder'></div>
```

11. Save the file.
12. Build the `DeployApp` project.
13. Deploy and run the `DeployApp` project. This should launch a web browser.
14. From your web browser, launch and test the `TextViewer` application as a Java Web Start application. Because the application is signed, it can access local resources.
15. Close the `DeployApp` project.

Practices for Lesson 14: Developing Secure Applications

Chapter 14

Practices for Lesson 14

Practices Overview

There are no practices for this lesson.

Practices for Lesson 15: Signing an Application and Authentication

Chapter 15

Practices for Lesson 15

Practices Overview

In these practices, you will sign your Java applications packaged in a `.jar` file. You will use both command line tools and NetBeans to sign your applications.

Practice 15-1: (Summary Level) Signing an Application by Using `keytool` and `jarsigner`

Overview

In this practice, sign an application by using the `keytool` and `jarsigner` utilities, and then deploy your application by using Java Web Start.

Assumptions

You have listened to the lecture portion of this lesson.

Summary

To sign an application, you need a pair of encryption keys. First, you will create a public and private key by using the `keytool` utility. With encryption keys in hand, you can build the `TextViewer` application and use the JAR file created to self-sign your application. You will deploy your signed application on a GlassFish server by using Java Web Start. Finally, test the application by using a browser to ensure that you are asked whether the application has access to the local file system. Then make sure that you can actually load local files and that your application is no longer stuck in the sandbox.

Tasks

1. Open the `TextViewer` project in the `practice01` directory for Lesson 15.
2. Build the `TextViewer` project.
3. Close the `TextViewer` project.
4. Copy the contents of the `dist` directory to the `TextViewerApp` directory in the `practice01` directory. Note that this includes all three files and the `web-files` directory.
5. Open a command prompt window.
6. Change into the `TextViewerApp` directory under `practice01`.
7. Use the `keytool` utility to create a keystore for your public and private keys.

   Note: The following steps assume that `keytool` is in your path. If it is not, you must add the *JDK*`\bin` or the *JRE*`\bin` directory to the path.

   - Enter the following command to create the keystore:

     ```
     keytool -genkey -alias mystorealias -keystore mystore
     ```

8. Enter the following information to complete the creation of your keystore. Your responses are highlighted in bold text.

   ```
   Enter keystore password: oracle
   Re-enter new password: oracle
   What is your first and last name? [Unknown]: John Adams
   What is the name of your organizational unit? [Unknown]:
   example.com
   ```

```
What is the name of your organization? [Unknown]: example.com
What is the name of your City or Locality? [Unknown]: Redwood
City
What is the name of your State or Province? [Unknown]: CA
What is the two-letter country code for this unit? [Unknown]: US
Is CN=John Adams, OU=example.com, O=example.com, L=Redwood City,
ST=CA, C=US correct?
[no]: yes
Enter key password for <mystorealias>
(RETURN if same as keystore password):
```

That should complete the creation of your public and private key and put them in a keystore file. If you look in the `TextViewerApp` directory, you should see a file named `mystore`.

9. Sign the JAR file by using the keystore file that you just created. Issue the following command:

   ```
   jarsigner -keystore mystore -storepass oracle -signedjar
   TextViewerSigned.jar TextViewer.jar mystorealias
   ```

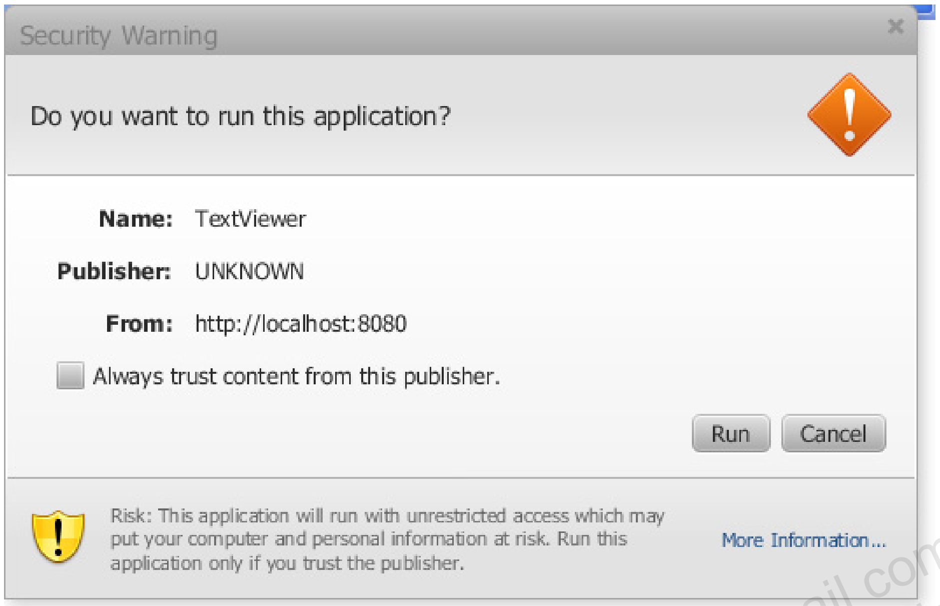
   This should create a second `.jar` file that is signed. Check the current directory to make sure that this is the case.

10. Edit your `.jnlp` file to request full permissions for the application.
    - Add the following security information after the resources tags in the file:

    ```
    <security>
      <all-permissions/>
    </security>
    ```

11. Remove any applet-related tags from your `.jnlp` file, because this is a Java Web Start–only deployment.
12. Change the `<jar>` element's `href` attribute to `TextViewerSigned.jar`.
13. Edit the `.html` file. Remove the embedded `jnlp_content` entries.
14. Edit the `.html` file. Remove the javafxEmbed()-related `<script>` information so that Java Web Start is the only option.
15. Copy the `TextViewerApp` directory into the Web directory of the `DeploymentApp` project.
16. Open the `DeploymentApp` project in NetBeans.
17. Edit `index.jsp` in the Web directory and create a link to the `TextViewerApp/TextViewer.html` file.
18. Deploy and run the application.

19. Verify that when you launch the application you get a confirmation dialog.


Security Warning

Do you want to run this application?

Name: TextViewer

Publisher: UNKNOWN

From: http://localhost:8080

Always trust content from this publisher.

Risk: This application will run with unrestricted access which may put your computer and personal information at risk. Run this application only if you trust the publisher.

20. Verify that you can open a local file.

Practice 15-1: (Detail Level) Signing an Application by Using `keytool` and `jarsigner`

Overview

In this practice, sign an application by using the `keytool` and `jarsigner` utilities, and then deploy your application by using Java Web Start.

Assumptions

You have listened to the lecture portion of this lesson.

Summary

To sign an application, you need a pair of encryption keys. First, you will create a public and private key by using the `keytool` utility. With encryption keys in hand, you can build the `TextViewer` application and use the JAR file created to self-sign your application. You will deploy your signed application on a GlassFish server by using Java Web Start. Finally, test the application by using a browser to ensure that you are asked whether the application has access to the local file system. Then make sure that you can actually load local files and that your application is no longer stuck in the sandbox.

Tasks

1. Open the `TextViewer` project in the `practice01` directory for Lesson 15.
2. Build the `TextViewer` project.
3. Close the `TextViewer` project.
4. Using File Explorer, navigate to the `dist` directory of the project and make sure that the `.jar`, `.jnlp`, and `.html` files have been generated for the project. Then, copy the contents of the `dist` directory to the `TextViewerApp` directory in the `practice01` directory. Note that this includes all three files and the `web-files` directory.
5. Open a command prompt window.
6. Change into the `TextViewerApp` directory under `practice01`.
7. Use the `keytool` utility to create a keystore for your public and private keys.

   Note: The following steps assume that `keytool` is in your path. If it is not, you must add the *JDK*`\bin` or the *JRE*`\bin` directory to the path.

   - Enter the following command to create the keystore:

     ```
     keytool -genkey -alias mystorealias -keystore mystore
     ```

8. Enter the following information to complete the creation of your keystore. Your responses are highlighted in bold text.

```
Enter keystore password: oracle
Re-enter new password: oracle
What is your first and last name? [Unknown]: John Adams
What is the name of your organizational unit? [Unknown]:
example.com
What is the name of your organization? [Unknown]: example.com
What is the name of your City or Locality? [Unknown]: Redwood
City
What is the name of your State or Province? [Unknown]: CA
What is the two-letter country code for this unit? [Unknown]: US
Is CN=John Adams, OU=example.com, O=example.com, L=Redwood City,
ST=CA, C=US correct?
[no]: yes
Enter key password for <mystorealias>
(RETURN if same as keystore password):
```

That should complete the creation of your public and private key and put them in a keystore file. If you look in the `TextViewerApp` directory, you should see a file named `mystore`.

9. Sign the JAR file by using the keystore file that you just created. Issue the following: command:

```
jarsigner -keystore mystore -storepass oracle -signedjar
TextViewerSigned.jar TextViewer.jar mystorealias
```

This should create a second `.jar` file that is signed. Check the current directory to make sure that this is the case.

10. Edit your `.jnlp` file to request full permissions for the application.
    - Add the following security information after the resources tags in the file:

```
<security>
  <all-permissions/>
</security>
```

11. Remove any applet-related tags from your `.jnlp` file, because this is a Java Web Start–only deployment.

```
<applet-desc  width="800" height="600" main-
class="com.javafx.main.NoJavaFXFallback"  name="TextViewer" />
```

12. Change the `<jar>` element's `href` attribute to `TextViewerSigned.jar`.

```
<jar href="TextViewerSigned.jar" size="19428" download="eager"
/>
```

13. Edit the `.html` file. Remove the embedded `jnlp_content` entries.

```
<script>
    function launchApplication(jnlpfile) {
        dtjava.launch(            {
                url : 'TextViewer.jnlp'
            },
            {
                javafx : '2.0+'
            },
            {}
        );
        return false;
    }
</script>
```

14. Edit the `.html` file. Remove the javafxEmbed()-related `<script>` information so that Java Web Start is the only option.

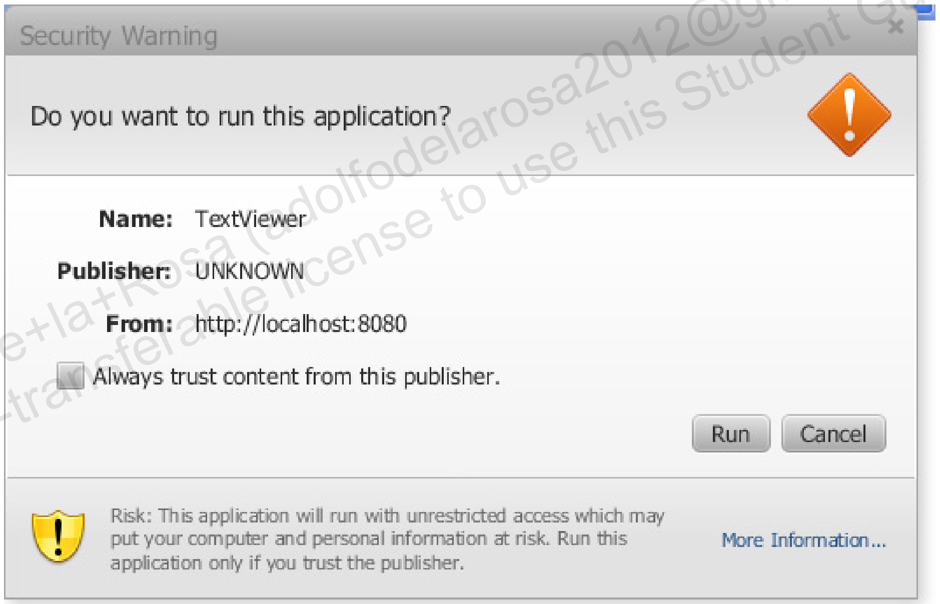
```
<script>
    function javafxEmbed() {
        dtjava.embed(
            {
                url : 'TextViewer.jnlp',
                placeholder : 'javafx-app-placeholder',
                width : 800,
                height : 600,
                jnlp_content :
'PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0idXRmLTgiPz4NCjxqbmxwIHN
wZWM9IjEuMCIgeG1sbnM6amZ4PSJodHRwOi8vamF2YWZ4LmNvbSIgaHJlZj0iVGV
4dFZpZXdlci5qbmxwIj4NCiAgPGluZm9ybWF0aW9uPg0KICAgIDx0aXRsZT5UZXh
0Vmlld2VyPC90aXRsZT4NCiAgICA8dmVuZG9yPm13MTE5MjU8L3ZlbmRvcj4NCiA
gICA8ZGVzY3JpcHRpb24+U2FtcGxlIEphdmFGWCAyLjAgYXBwbGljYXRpb24uPC9
kZXNjcmlwdGlvbj4NCiAgICA8b2ZmbGluZS1hbGxvd2VkLz4NCiAgPC9pbmZvcm1
hdGlvbj4NCiAgPHJlc291cmNlcyBvcz0iV2luZG93cyI+DQogICAgPGpmeDpqYXZ
hZngtcnVudGltZSB2ZXJzaW9uPSIyLjArIiBocmVmPSJodHRwOi8vamF2YWRsLnN
1bi5jb20vd2ViYXBwcy9kb3dubG9hZC9HZXRGaWxlL2phdmFmeC1sYXRlc3Qvd2l
uZG93cy1pNTg2L2phdmFmeDIuam5scCIvPg0KICA8L3Jlc291cmNlcz4NCiAgPHJ
lc291cmNlcz4NCiAgICA8ajJzZSB2ZXJzaW9uPSIxLjYrIiBocmVmPSJodHRwOi8
vamF2YS5zdW4uY29tL3Byb2R1Y3RzL2F1dG9kbC9qMnNlIi8+DQogICAgPGphciB
ocmVmPSJUZXh0Vmlld2VyLmphciIgc2l6ZT0iMTc0MzEiIGRvd25sb2FkPSJlYWd
lciIgLz4NCiAgPC9yZXNvdXJjZXM+DQogIDxhcHBsZXQtZGVzYyAgd2lkdGg9Ijg
wMCIgaGVpZ2h0PSI2MDAiIG1haW4tY2xhc3M9ImNvbS5qYXZhZngubWFpbi5Ob0p
hdmFGWEZhbGxiYWNrIiAgbmFtZT0iVGV4dFZpZXdlciIgLz4NCiAgPGpmeDpqYXZ
hZngtZGVzYyAgd2lkdGg9IjgwMCIgaGVpZ2h0PSI2MDAiIG1haW4tY2xhc3M9ImN
vbS5leGFtcGxlLmd1aS5UZXh0Vmlld2VyIiAgbmFtZT0iVGV4dFZpZXdlciIgLz4
NCiAgPHVwZGF0ZSBjaGVjaz0iYmFja2dyb3VuZCIvPg0KPC9qbmxwPg0K'
            },
            {
```

```
                        javafx : '2.0+'
                    },
                    {}
                );
            }
            <!-- Embed FX application into web page once page is loaded
        -->
            dtjava.addOnloadCallback(javafxEmbed);
        </script>
```

15. Copy the `TextViewerApp` directory into the web directory of the `DeploymentApp` project.
16. Open the `DeploymentApp` project in NetBeans.
17. Edit `index.jsp` in the web directory and create a link to the `TextViewerApp/TextViewer.html` file.

```
        <a href="TextViewerApp/TextViewer.html">Text Viewer Signed</a>
```

18. Deploy and run the application.
19. Verify that when you launch the application you get a confirmation dialog.


Security Warning

Do you want to run this application?

Name: TextViewer

Publisher: UNKNOWN

From: http://localhost:8080

Always trust content from this publisher.

Run Cancel

Risk: This application will run with unrestricted access which may put your computer and personal information at risk. Run this application only if you trust the publisher.

More Information...

20. Verify that you can open a local file.

Practice 15-2: (Summary Level) Signing and Deploying `TextViewer` by Using NetBeans

Overview

In this practice, sign the `TextViewer` application by using NetBeans.

Assumptions

You have listened to the lecture portion of this lesson and have completed prior practices.

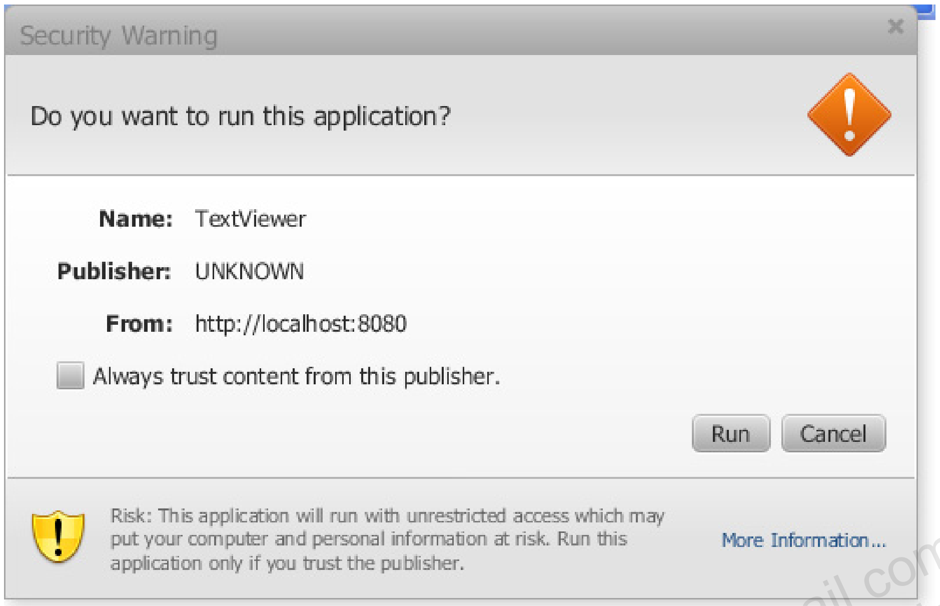
Summary

In this practice, let NetBeans do the heavy lifting. Modify your project so that NetBeans will create a self-signed application for you. Just modify your project properties and the rest is automatic.

Tasks

1. In the `practice02` directory for this lesson, open the `TextViewer` project. Alternatively, continue to work with the `TextViewer` project from the previous lesson.
2. Edit the project properties to allow your application to run with unrestricted access. Configuration your project to self-sign your output JAR file.
3. Clean and build the project.
4. Open the `DeploymentApp` project.
5. Create a directory named `TextViewerApp` under the `web` directory.
6. Copy the contents of the `dist` directory to the `DeploymentApp` project.
7. Open the `DeploymentApp` project.
8. Edit the `index.jsp` and create a link to your application as you did in the previous practice.
9. Deploy and run the application.

10. Verify that, when you launch the application, you get a confirmation dialog.



Security Warning

Do you want to run this application?

Name: TextViewer

Publisher: UNKNOWN

From: http://localhost:8080

☐ Always trust content from this publisher.

Risk: This application will run with unrestricted access which may put your computer and personal information at risk. Run this application only if you trust the publisher.

11. Verify that you can open a local file.

Practice 15-2: (Detail Level) Signing and Deploying `TextViewer` by Using NetBeans

Overview

In this practice, sign the `TextViewer` application by using NetBeans.

Assumptions

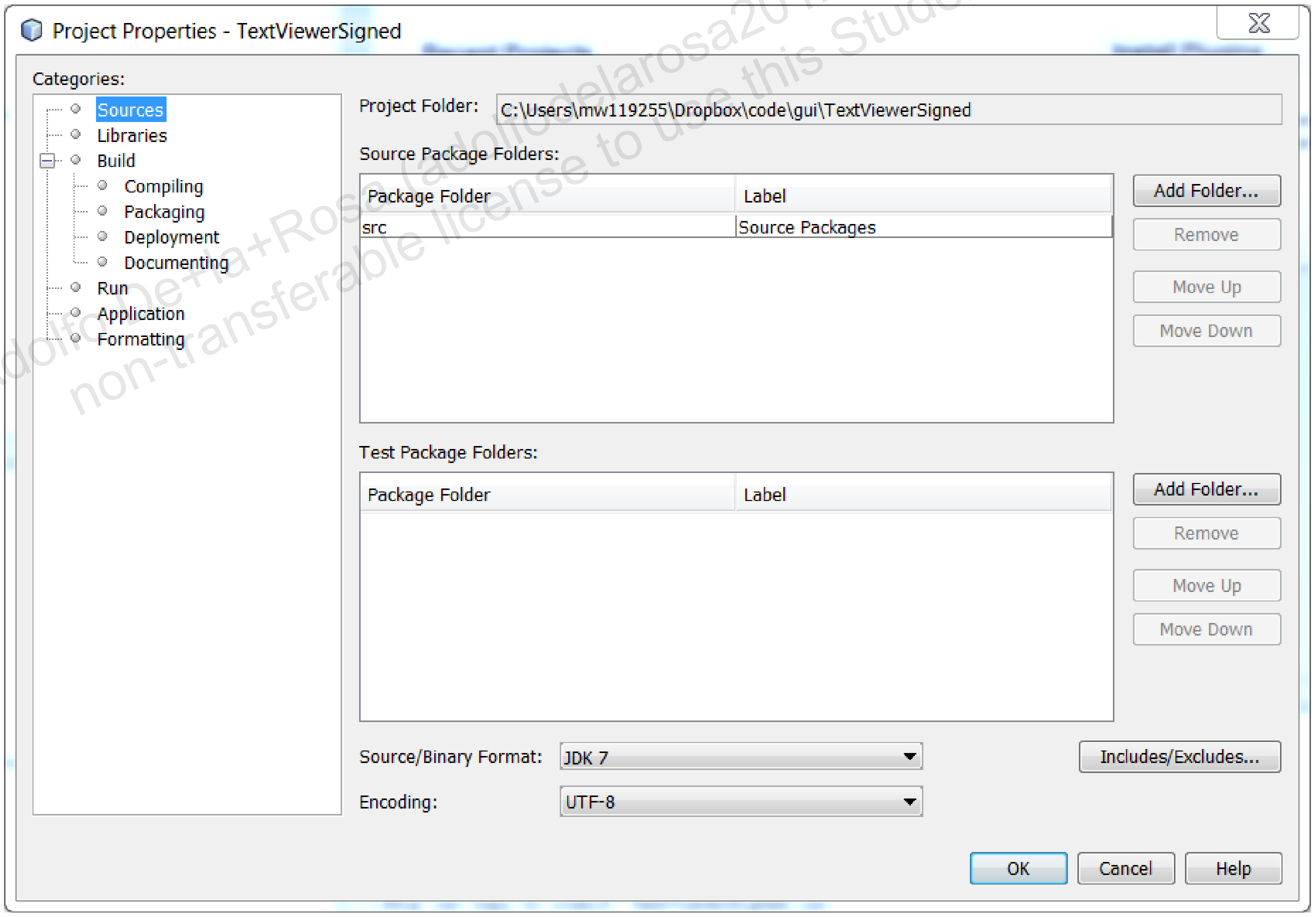
You have listened to the lecture portion of this lesson and have completed prior practices.

Summary

In this practice, let NetBeans do the heavy lifting. Modify your project so that NetBeans will create a self-signed application for you. Just modify your project properties and the rest is automatic.

Tasks

1. In the `practice02` directory for this lesson, open the `TextViewer` project. Alternatively, continue to work with the `TextViewer` project from the previous lesson.
2. Right-click the project name and choose properties.



Project Folder: C:\Users\mw119255\Dropbox\code\gui\TextViewerSigned

Source Package Folders:

Package Folder	Label
src	Source Packages

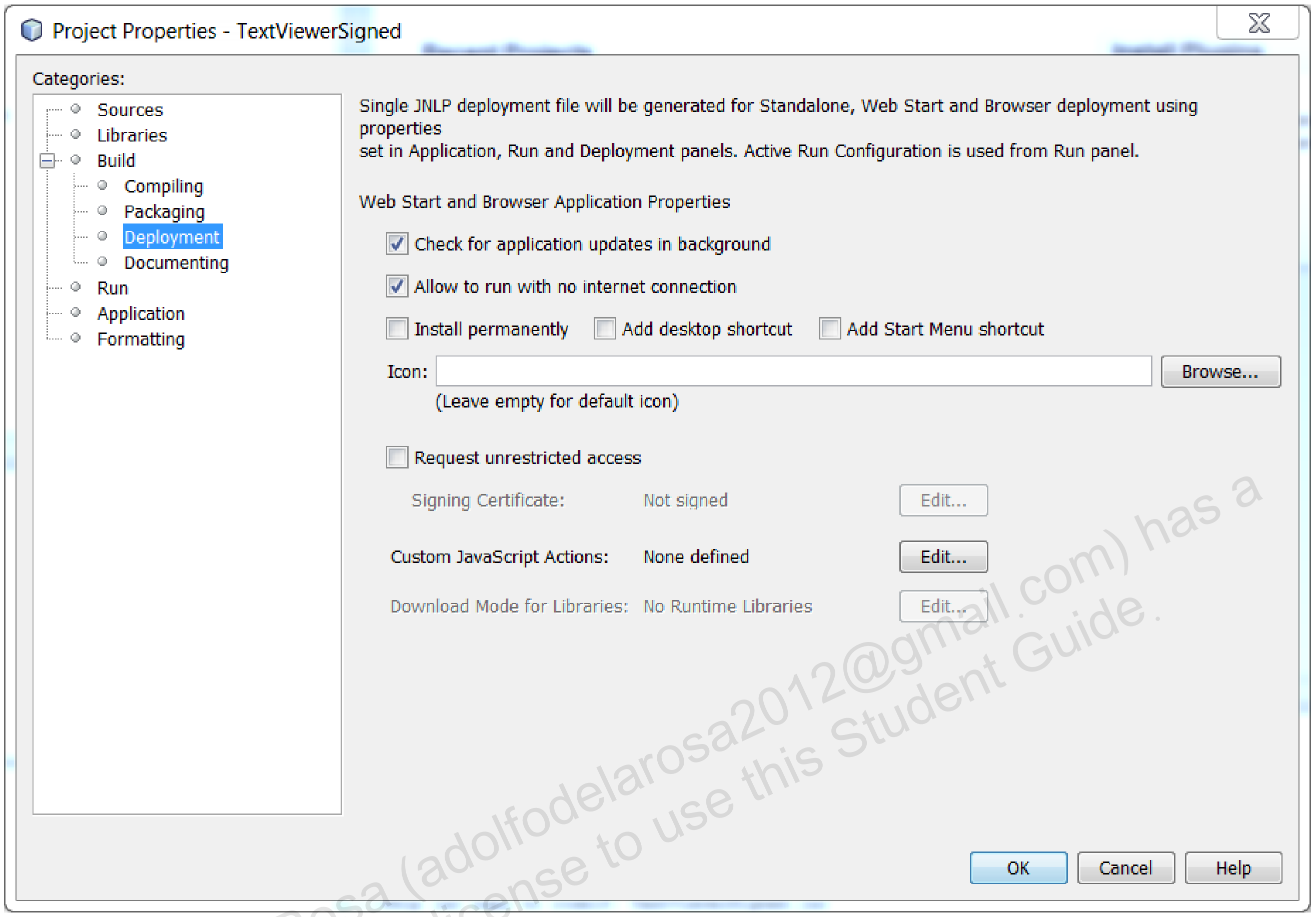
Test Package Folders:

Package Folder	Label

Source/Binary Format: JDK 7

Encoding: UTF-8

3. Click the Deployment option under build.



Project Properties - TextViewerSigned

Categories:

Single JNLP deployment file will be generated for Standalone, Web Start and Browser deployment using properties
set in Application, Run and Deployment panels. Active Run Configuration is used from Run panel.

Web Start and Browser Application Properties

- [✓] Check for application updates in background
- [✓] Allow to run with no internet connection
- [ ] Install permanently [ ] Add desktop shortcut [ ] Add Start Menu shortcut

Icon:

(Leave empty for default icon)

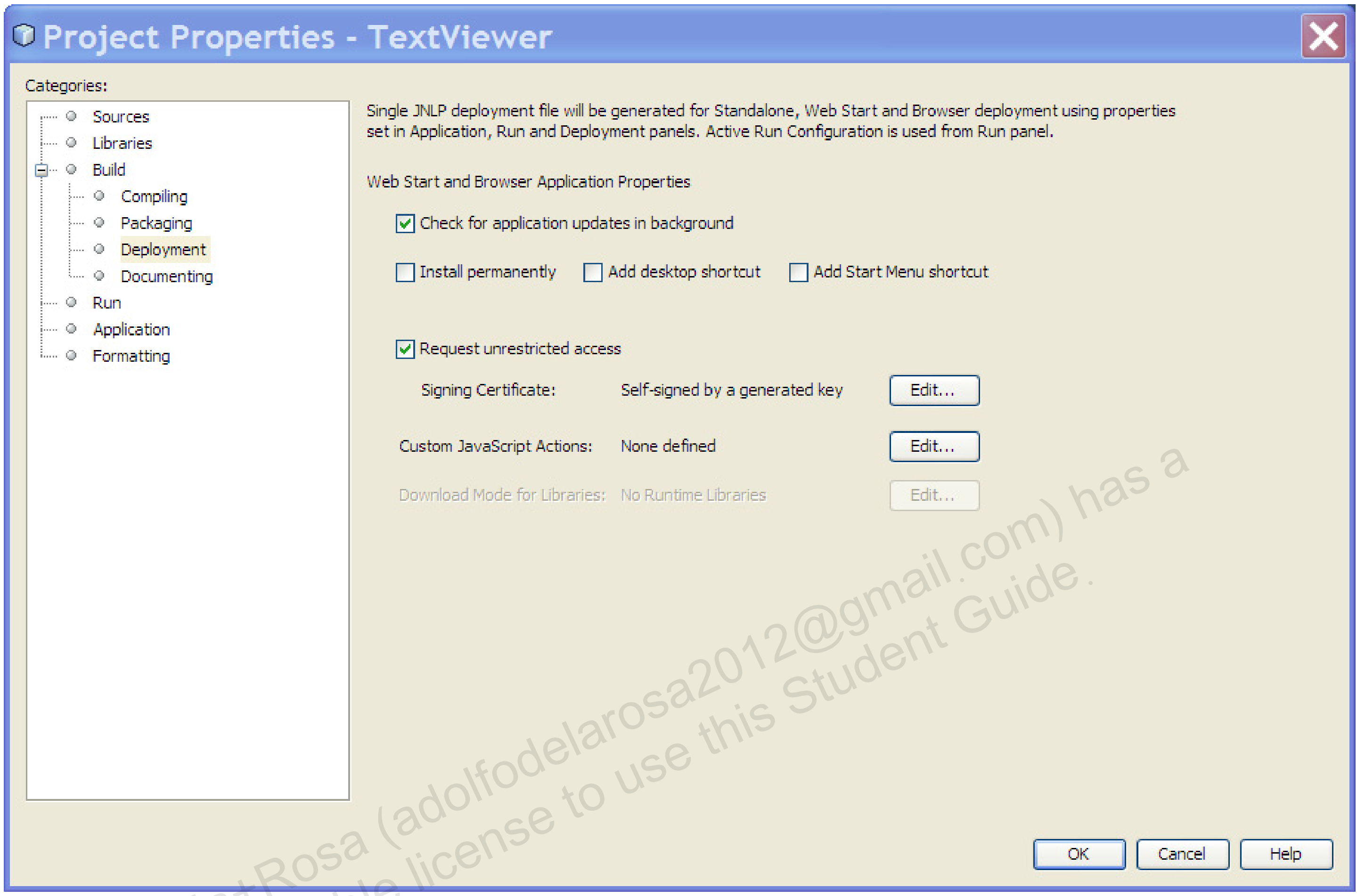
- [ ] Request unrestricted access

Signing Certificate: Not signed

Custom JavaScript Actions: None defined

Download Mode for Libraries: No Runtime Libraries

4. Check Request unrestricted access to self-sign the application.



Single JNLP deployment file will be generated for Standalone, Web Start and Browser deployment using properties set in Application, Run and Deployment panels. Active Run Configuration is used from Run panel.

Web Start and Browser Application Properties

[✓] Check for application updates in background

[ ] Install permanently [ ] Add desktop shortcut [ ] Add Start Menu shortcut

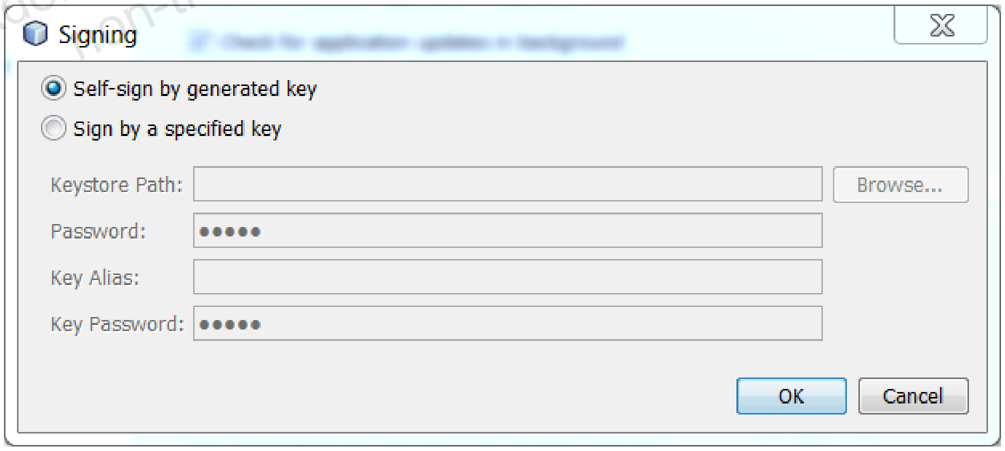
[✓] Request unrestricted access

Signing Certificate: Self-signed by a generated key Edit...

Custom JavaScript Actions: None defined Edit...

Download Mode for Libraries: No Runtime Libraries Edit...

5. If you click the Edit button by the “Self-signed by a generated” option, you can choose an auto-generated signing certificate.



[● Self-sign by generated key]

○ Sign by a specified key

Keystore Path:

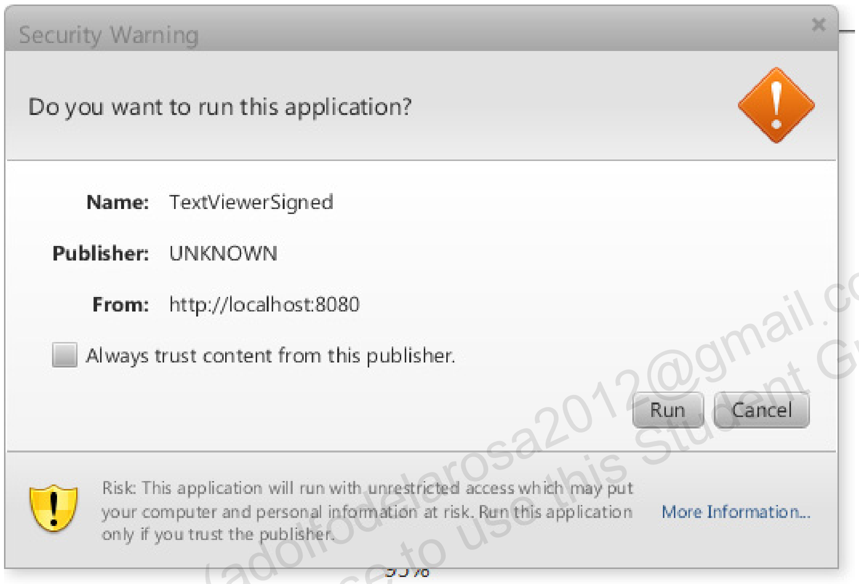
Password: ●●●●●

Key Alias:

Key Password: ●●●●●

6. Click OK and OK again to save the changes to the project.
7. Clean and build the project.
8. Open the `DeploymentApp` project.

9. Create a directory named `TextViewerApp` under the `web` directory.
10. Copy the contents of the `dist` directory to in the `DeploymentApp` project.
11. Open the `DeploymentApp` project.
12. Edit the `index.jsp` and create a link to your application as you did in the previous practice.
13. Deploy and run the application.
14. Verify that when you launch the application you get a confirmation dialog.



Security Warning

Do you want to run this application?

Name: TextViewerSigned

Publisher: UNKNOWN

From: http://localhost:8080

☐ Always trust content from this publisher.

Risk: This application will run with unrestricted access which may put your computer and personal information at risk. Run this application only if you trust the publisher.

15. Verify that you can open a local file.

Practices for Lesson 16: Logging

Chapter 16

Practices for Lesson 16

Practices Overview

In these practices, you will learn about the following:

- Logging APIs and the `Logger` class methods of `java.util.Logger` package
- Adding logging to Java applications

Practice 16-1: (Summary Level) Logging in Java Applications

Overview

In this practice, you will add logging for a Java application. Note that practices for this chapter do not use JavaFX and there is no GUI.

You have to implement logging in a Java application, `Account.java`. Logging has to be implemented for various banking transactions to a file, `AccountInfo.log`, and exceptions in an `Exception.log` file.

Assumptions

NetBeans is running.

Tasks

1. Open the `BankTransactions` project in NetBeans, located in the `D:\labs\16-Logging\practices\practice01` folder.
2. Expand the project folders and open `Account.java` in the code editor window.
3. Instantiate the Logger component, and add the following code to `Account.java`:

```
Logger logger =
Logger.getLogger("com.logging.practice01.Account");
 try {
       logger.setLevel(Level.INFO);
       logger.addHandler(newFileHandler("AccountInfo.log"));
      } catch (IOException ex) {
      }
```

4. Create an account and perform the following steps:
   - Log the information to a file called `AccountsInfo.log`.
   - Set the Log Level to `info`.
   - Log the message: "Account is created ".
5. Deposit an amount in the account and perform the following steps:
   - Deposit an amount to the account by invoking the `deposit(double amount)` method.
   - Log the information to a file called `AccountsInfo.log`.
   - Set the Log Level to `info`.
   - Log the message: "Money is deposited".
6. Withdraw an amount from the account and perform the following steps:
   - Withdraw an amount by invoking the withdraw method in the `main()` method.
   - Log the information to a file called `AccountsInfo.log`.
   - Set the Log Level to `info`.
   - Log the message: "Money is withdrawn”.

7. Withdraw an amount greater than the available balance and perform the following steps:
   - Throw an exception. The exception details should be logged.
   - Log the information to a file called `ExceptionInfo.log`.
   - Set the Log Level to SEVERE.
   - Log the message: "Insufficient Funds".
8. Deposit an amount less than or equal to zero and perform the following steps:
   - Throw an exception. The exception details should be logged.
   - Log the information to a file called `ExceptionInfo.log.`
   - Set the formatter type to `SimpleFormatter`.
   - Set the Log Level to SEVERE.
   - Log the message: "deposit value is negative".
9. Run the project.
10. To verify the output, perform the following steps:
    a. Browse to the `D:\labs\16-Logging\practices\practice01\` folder.
    b. Examine the contents of two files: `AccountInfo.log` and `Exception.log`.
    c. Note the logging details that were updated in the files.

Practice 16-1: (Detail Level) Logging in Java Applications

Overview

In this practice, you will add logging for a Java application. Note that practices for this lesson do not use JavaFX and there is no GUI.

You have to implement logging in a Java application, `Account.java`. Logging has to be implemented for various banking transactions to a file, `AccountInfo.log`, and exceptions in an `Exception.log` file.

Assumptions

NetBeans is running.

Tasks

1. In NetBeans, complete the following steps to open the `BankTransactions` project:
   a. Select File > Open Project.
   b. In the Open Project dialog box, browse to the `D:\labs\16-Logging\practices\practice01` folder.
   c. Select the `BankTransactions` project and click the Open Project button.
2. In the Projects window, expand `BankTransactions` > *`Source Packages >`* `com.logging.practice01.`
3. Click `Account.java` to open it in the code editor window.
4. Add a `main()` method to the class, `Account.java.`

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

5. To instantiate the Logger component, add the following code to the `main()` method:

```
Logger logger =
Logger.getLogger("com.logging.practice01.Account");
 try {
       logger.setLevel(Level.INFO);
       logger.addHandler(newFileHandler("AccountInfo.log"));
      } catch (IOException ex) {
      }
```

The following steps demonstrate various scenarios for enabling logging in `Account.java`.

Complete the following steps:

6. Scenario 1 for logging: To create an account and log the following information:
   - Log the information to a file called `AccountsInfo.log.`
   - Set the Log Level to `info`.
   - Log the message: "Account is created".

   a. To create an account, add the following statement to the `main()` method:

```
Account James = new Account("James", 123456, 1600.50);
```

   b. To log this information in `AccountsInfo.log`, write the following statement:

```
logger.log(Level.INFO," Account is created",James);
```

Because we are passing the James object to the log method, the Account details of the James object is logged to the AccountsInfo.log file.The Account details of James object is also logged to the AccountsInfo.log file.

7. Scenario 2 for logging: Deposit an amount in the account:
   - Deposit an amount to the account by invoking the deposit(double amount) method.
   - Log the information to a file called AccountsInfo.log.
   - Set the Log Level to info.
   - Log the message:"Money is deposited".

   To implement logging for the above scenario, perform the following steps:

   a. To deposit an amount to the account, invoke the deposit() method in the main() method.

   ```
   James.deposit(450.36);
   ```

   b. To log this information in AccountsInfo.log, write the following statement in the main() method:

   ```
    logger.log( Level.INFO," Money is deposited");
   ```

   c. To verify the balance after deposit transaction, add the following two statements to main() method:

   ```
   double depositbalance=James.getBalance();
   System.out.println(" Balance after withdrawal is " +
   depositbalance);
   ```

8. Scenario 3 for logging: Withdraw an amount from the account:
   - Withdraw an amount by invoking the withdraw method in the main() method.
   - Log the information to a file called AccountsInfo.log.
   - Set the Log Level to info.
   - Log the message:"Money is withdrawn".

   To implement logging for the above scenario, perform the following steps:

   a. To withdraw an amount to the account, invoke the withdraw() method in the main() method.

   ```
   James.withdraw(550, 25);
   ```

   b. To log this information in AccountsInfo.log, write the following statement in the main() method:

   ```
    logger.log(Level.INFO,"Money is withdrawn");
   ```

   c. To verify the balance after the deposit transaction, add the following two statements to main() method:

   ```
   double withdrawbalance=James.getBalance();
    System.out.println(" Balance after withdrawal is " +
   withdrawbalance);
   ```

9. Scenario 4: Withdraw an amount greater than the balance amount available:
   - It throws an exception. The exception details should be logged.
   - Log the information to a file called `ExceptionInfo.log`.
   - Set the Log Level to SEVERE.
   - Log the message:"Insufficient Funds".

   To implement logging for the above scenario, perform the following steps:

   a. To withdraw an amount greater than the balance amount. Add the following statement to the `main()` method:

   ```
   double withdraw = James.withdraw(5000, 25);
   ```

   b. Because this is not a valid operation, it throws an exception called "Insufficient Balance". To log this exception in the `Exception.log` file, add the following code as part of exception handling:

   ```
   catch (Exception ex) {
     try {

        logger.addHandler(new FileHandler("Exception.log"));
        }
   catch (IOException ex1) {
      }
      logger.log(Level.SEVERE, "Insufficient Funds");

      }
   ```

10. Scenario 5: Deposit an amount lesser than or equal to zero:
    - It throws an exception. The exception details should be logged.
    - Log the information to a file called ExceptionInfo.log.
    - Set the formatter type to SimpleFormatter.
    - Set the Log Level to SEVERE.
    - Log the message:"deposit value is negative".

    To implement logging for the above scenario, perform the following steps:

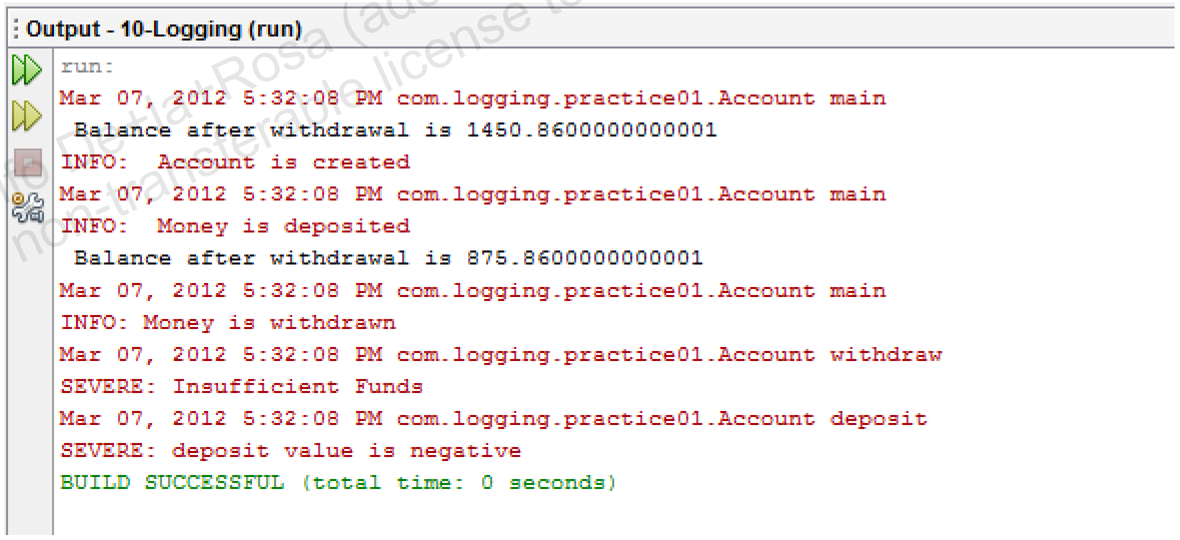
    a. To deposit an amount less than or equal to zero. Add the following statement to the `main()` method:

    ```
    James.deposit(0.0);
    ```

b. Examine the `deposit()` method. Because it is not a valid exception, it throws an `Exception`. To log this exception in the `Exception.log` file, add the following code as part of exception handling:

```
 catch (Exception ex) {
            try {
                FileHandler fh=new FileHandler("Exception.log");
                logger.addHandler(fh);
                fh.setFormatter(new SimpleFormatter());
            } catch (IOException ex1) {
            }
            logger.log(Level.SEVERE, "deposit value is 
negative");

        }
```

11. Save the file and build the project. Remove any compilation errors if any.
12. In the Projects window, right-click `Account.java` and select Run File from the right-click menu.
13. To verify the output, perform the following steps:
    a. Browse to the `D:\labs\16-Logging\practices\practice01\` folder.
    b. Examine the contents the two files: `AccountInfo.log` and `Exception.log`.
    c. Examine the logging details updated in the files.
14. You see the following output in the console:



```
run:
Mar 07, 2012 5:32:08 PM com.logging.practice01.Account main
 Balance after withdrawal is 1450.8600000000001
INFO:  Account is created
Mar 07, 2012 5:32:08 PM com.logging.practice01.Account main
INFO:  Money is deposited
 Balance after withdrawal is 875.8600000000001
Mar 07, 2012 5:32:08 PM com.logging.practice01.Account main
INFO: Money is withdrawn
Mar 07, 2012 5:32:08 PM com.logging.practice01.Account withdraw
SEVERE: Insufficient Funds
Mar 07, 2012 5:32:08 PM com.logging.practice01.Account deposit
SEVERE: deposit value is negative
BUILD SUCCESSFUL (total time: 0 seconds)
```

Practices for Lesson 17: Implementing Unit Testing and Using Version Control

Chapter 17

Practices for Lesson 17

Practices Overview

In these practices, you will:

- Create and execute a test case
- Set up a test fixture
- Perform a parameterized test

Practice 17-1: (Summary Level) Creating and Executing Test Cases

Overview

In this practice, you will learn how to:

- Write a test case
- Set up a test fixture
- Execute a test case

Assumptions

NetBeans is running on your system.

Tasks

1. In NetBeans, open the `17JUnitPractices` project located in the `D:\labs\17-JUnit` folder.
2. Install the JUnit library: Add `junit-4.10.jar` located at `D:\labs\resources\setup` to the project properties.
3. Expand the project folder and open `Stack.java`.
4. Examine `Stack.java` to understand the working of Stack.
5. In the Projects window, expand packages and open `StackTest.java` in the code editor window.
6. Define a class-level member variable, `stack`, object of type Stack.
7. Create an instance of the Stack class, `stack`, in the `setUp()` method.

   ```
   stack=new Stack();
   ```

8. In the `tearDown()` method, set the stack instance `stack` to `null`. This destroys the Stack instance.

   ```
   stack=null;
   ```

9. Write a JUnit test case, `testpush()`, for the `push()` method. To the `testpush()` method, do the following:
   a. Add three elements to the stack.
   b. Invoke the `getSize()` method to know the size of the stack.
   c. Using Assert, compare whether the actual size of the stack and the expected size are the same.
   d. This test case is “pass” if the Assert statement returns `true` indicating that the `push()` method has added three elements.

10. Create test case `testPop()` for the `pop()` method of the `Stack` class.
    a. Delete the IDE-generated code for the method, `testPop()`.
    b. Add the following lines of code in the `testPop()` method to do the following:
        1) Add two elements to the stack.
        2) Write an `if` statement to check whether it is an empty stack.
        3) Add two Assert statements to verify the `pop()` method:
            - Invoke the `pop()` method to compare whether the deleted element is true.
            - Invoke the `peek()` method to compare whether the element on top of the stack is 1.
11. Run the project.
12. Verify whether all the test cases are "pass." The Junit bar in the output console will be green if the test cases are "pass"; otherwise, it will be in red.

Practice 17-1: (Detailed Level) Creating and Executing Test Cases

Overview

In this practice, you will learn how to:

- Write a test case
- Set up a test fixture
- Execute a test case

Assumptions

NetBeans is running on your system.

Tasks

1. In NetBeans, complete the following steps to open the `17JUnitPractices` project:
   a. Select File > Open Project.
   b. In the Open Project dialog box, browse to the `D:\labs\17-JUnit\Practices` folder.
   c. Select the `17JUnitPractices` project and click the Open Project button.

   Note that you could observe compilation errors in the project, because the JUnit library is not added.
2. Complete the following steps to install the JUnit library:
   a. Right-click the `17JUnitPractices` project and select the Properties menu option.
   b. Select Libraries from the Categories section, and click Add JAR/Folder.
   c. Browse to the `D:\labs\resources\setup` directory.
   d. Select `junit-4.10.jar` and click Open.
   e. Click OK to exit the wizard.
3. In the Projects window, expand `17JUnitPractices > Source Packages > lesson17.practice1`.
4. Double-click `Stack.java` to open it in the code editor window.
5. Examine `Stack.java` to understand the working of Stack.
6. Preparing JUnit Test Cases for `Stack.java`:
   a. Right-click the package `lesson17.practice1` in the project pane and select `New > Other > JUnit > JUnit Test`.
   b. Name the class as `StackTest` and package as `lesson17.practice1`.
   c. In the Projects window, expand `17JUnitPractices > Test Packages > lesson17.practice1`.
   d. Double-click `StackTest.java` to open it in the code editor window.

7. Define a class level member variable, `stack`, object of type Stack.

```
Stack stack;
```

8. To create a test fixture, modify the `setUp()` and `tearDown()` methods as follows .

   a. Write the following code in the `setUp()` method. This creates an instance of Stack class.

   ```
   stack=new Stack();
   ```

   `setUp()` will be invoked during execution of each test case.

   b. Write the below code in the `tearDown()` method. This destroys the Stack instance.

   ```
   stack=null;
   ```

   `tearDown()` will be invoked after execution of each test case.

9. Create a test case, `testPush()`, for the `push()` method of the `Stack` class. In JUnit 4, a test method does not have to be prefixed with `test` but instead uses the @Test annotation.

   a. Add a method code, `testPush()`.

   ```
   @Test
       public void testPush() {
   }
   ```

   b. Add code in the `testpush()` method to do the following:

      1) Add three elements to the stack.

      2) Invoke the `getSize()` method to know the size of the stack.

      3) Using Assert, compare whether the actual size of the stack and the expected size are the same.

   This test case is "pass" if the Assert statement returns `true` indicating that the `push()` method has added three elements.

   ```
           stack.push(10);
           stack.push(11);
           stack.push(12);
           assertEquals(stack.getSize(), 3);
   ```

10. Create a test case, `testPop()`, for the `pop()` method of the `Stack` class.

   a. Add a `testPop()` method.

   ```
        @Test
       public void testPop() {
   }
   ```

   b. Add code in the `testPop()` method to do the following :

      1) Add two elements to the stack.

      2) Write an `if` statement to check whether it is an empty stack.

3) Add two Assert statements to verify the `pop()` method:
   - Invoke the `pop()` method to compare whether the deleted element is true.
   - Invoke the `peek()` method to compare whether the element on top of the stack is 1.

11. Create a test case, `testIsEmpty()` for the `isEmpty()` method of the Stack class.
    a. Add a method, `testIsEmpty()`.

```
@Test
    public void testIsEmpty() {
}
```

    b. Write an `assertTrue` statement that returns `true` if the stack is empty.

```
assertTrue(stack.isEmpty());
```

12. Create a test case, `testPeek()` for the `peek()` method of the Stack class.
    a. Add a method, `testPeek()`.

```
     @Test
    public void testPeek() {
}
```

    b. Add three elements to the stack.
    c. Write an Assert statement to verify the `peek()` method.
    d. Invoke the `peek()` method and compare whether the value on top of the stack is 3.

```
          stack.push(1);
          stack.push(2);
          stack.push(3);
          assertEquals(stack.peek(), 3);
```

13. Create a test case, `testGetSize()`, for the `getSize()` method of the Stack class.
    a. Add a method, `testGetSize()`.

```
   @Test
    public void testGetSize() {
}
```

    b. Add three elements to the stack.
    c. Write an `Assert` statement to verify the `getSize()` method.
    d. Invoke the `getSize()` method and compare whether the value on top of the stack is 3.

```

          stack.push(1);
          stack.push(2);
          stack.push(3);
          assertEquals(stack.getSize(), 3);
```

14. Run the project.
15. Verify whether all the test cases are “pass.” The Junit bar in the output console will be green if the test cases are “pass”; otherwise, it will be in red.

Practice 17-2: (Summary Level) Writing Test Cases Using Additional JUnit 4 Annotations

Overview

In this practice, you will be familiarized with additional JUnit 4 annotations to write effective test cases to perform Unit Testing.

Assumptions

NetBeans is running on your system.

Tasks

1. Open the `17JUnitPractices` project located in the `D:\labs\17-JUnit` folder.
2. In the Projects window, expand `17JUnitPractices > Source Packages > lesson17.practice2` and open `Calculator.java.`
3. Examine `Calculator.java` to understand the business logic.
4. In the Projects window, expand `17JUnitPractices > Test Packages > lesson17.practice2` and `CalculatorTest.java.`
5. Use `@Ignore` annotation to the `multiply()` test case to indicate that it is not fully implemented.
6. Modify the `@Test` annotation of `divideByZero()` method to indicate that division by zero throws an `ArithmeticException`.

```
@Test(expected = ArithmeticException.class)
```

7. Modify the `@Test` annotation of the `squareRoot()` method to indicate that the test fails because it takes longer than 200ms to compute the square root of the given number.

```
          @Test(timeout=200)
```

8. Run the project.
9. Verify the output. All the test cases will be “pass” except the `squareRoot()` test case.

Practice 17-2: (Detail Level) Writing Test Cases Using Additional JUnit 4 Annotations

Overview

In this practice, you will be familiarized with additional JUnit 4 annotations to write effective test cases to perform Unit Testing.

Assumptions

NetBeans is running on your system.

Tasks

1. In NetBeans, complete the following steps to open the `17JUnitPractices` project:
   a. Select File > Open Project.
   b. In the Open Project dialog box, browse to the `D:\labs\17-JUnit` folder.
   c. Select the `17JUnitPractices` project and click the Open Project button.
2. In the Projects window, expand `17JUnitPractices > Source Packages > lesson17.practice2.`
3. Double-click `Calculator.java` to open it in the code editor window.
4. Examine `Calculator.java` to understand the business logic.
5. In the Projects window, expand `17JUnitPractices > Test Packages > lesson17.practice2.`
6. Double-click `CalculatorTest.java` to open it in the code editor window. `CalculatorTest.java` is the unit Testing file generated for `Calculator.java.`
7. Add the following code to the `multiply()` test case:

```
@Ignore("not ready yet")
       @Test  public void multiply() {
     }
```

   The `@Ignore` annotation indicates that the test runner will not run these tests, and it will be skipped.

8. Modify the `@Test` annotation of the `divideByZero()` method as follows:

```
@Test(expected = ArithmeticException.class)
```

   The `divideByZero()` method tests that division by zero throws an `ArithmeticException`. The code indicates the use of an annotation to declare that the exception is expected. We can avoid using `try` and `catch` blocks.

9. Modify the `@Test` annotation of the `squareRoot()` method as follows:

   ```
   @Test(timeout=200)
   ```

   The `squareRoot()` method tests performance. The test fails because it takes longer than 200ms to compute the square root of the given number, because the `squareRoot()` method in `Calculator.java` contains an infinite loop.

10. Save the file and build the project. Remove any compilation errors if any.
11. In the Projects window, right-click `CalculatorTest.java` and select Run File from the right-click menu.
12. Verify the output. All the test cases will be "pass" except `squareRoot()` test case.

Practice 17-3: Creating a Test Suite

Overview

In this practice, you will create a Test Suite using JUnit 4. With JUnit 4 you use annotations to create a Test Suite. You will learn how to create a Test Suite using `@RunWith` and `@Suite` annotations.

Assumptions

NetBeans is running on your system.

Tasks

1. In NetBeans, complete the following steps to open the project:
   a. Select File > Open Project.
   b. In the Open Project dialog box, browse to the `D:\labs\17-JUnit folder`.
   c. Select `17JUnitPractices` project and click the Open Project button
2. In the Projects window, expand `17JUnitPractices > Test Packages`.
3. Right-click Test Package and select New > Java Class.
4. Save the *class* as `TestSuiteDemo` and *package name* as `lesson17.practice3`.
5. Double-click `TestSuiteDemo.java` in the project pane to open it in the code editor window.
6. Complete the following steps to write a program that creates a test suite.
   a. Import the following classes:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

   b. Add an `@RunWith` annotation. The `@RunWith` annotation tells JUnit to use the `org.junit.runner.Suite`. This runner allows you to manually build a test suite containing tests from many classes.

```
@RunWith(Suite.class)
```

   c. Add an `@Suite` annotation .The names of the classes that are part of the test suite are defined in `@Suite.SuiteClass`.

```
@Suite.SuiteClasses({
        CalculatorTest.class,
        StackTest.class
        })
```

   When you run this class, it will run test cases of both `CalculatorTest.java` and `StackTest.java`.
7. Save the file and build the project. Remove any compilation errors if any.
8. In the Projects window, right-click `TestSuiteDemo.java` and select Run File from the right-click menu.

9. Verify the output. All the test cases in `CalculatorTest.java` and `StackTest.java` should be executed.

Practice 17- 4: Performing Parameterized Testing

Overview

In this practice, you will learn how to perform Parameterized tests. JUnit 4 comes with a special runner, Parameterized, which allows you to run the same test with different data.

In order to do parameterized testing, you do the following:

- Mark test class as a parameterized test by using `@RunWith(Parameterized.class)`.
- The class must then have the following three entities:
    1. A static method that generates and returns test data
    2. A single constructor that stores the test data
    3. A test case

Assumptions

NetBeans is running on your system.

Tasks

1. In the NetBeans Projects window, expand `17JUnitPractices > Source Packages > lesson17.practice4`.
2. Select `StackParameterizedTest.java` to open it in the code editor window.
3. Complete the following steps to write a program that creates a parameterized test.
    a. Import the following classes:

```
import org.junit.runner.RunWith;
import org.junit.runners.*;
import org.junit.runners.Parameterized.Parameters;
import org.junit.Assert*.
import java.util.*;
```

    b. Add an `@RunWith(Parameterized.class)` annotation to the class.

```
@RunWith(value = Parameterized.class)
public class StackTest {
```

    c. Declare a class level variable, `number`, of the `int` type.

```
int number;
```

    d. Modify the constructor, `StackParametrizedTest()` as follows:

```
public StackTestParameterized(int number) {
   this.number = number;
}
```

e. Write the following code to create a static method, `data()` ,which generates and returns test data:

```
public static Collection data() {
   Object[][] data = new Object[][] { { 1 }, { 2 }, { 3 }, { 17
} };
   return Arrays.asList(data);
 }
```

f. Add `@Parameters` annotation to `the data()` method.

```
@Parameters
public static Collection data()
```

g. Modify the `testPush()` method to use parameterized testing .

```
    @Test
    public void testPush() {
       stack.push(number);
       assertEquals(stack.peek(),number);

    }
```

4. Save the file and build the project. Remove compilation errors, if any.
5. In the Projects window, right-click `StackParameterizedTest.java` and select Run File from the right-click menu.
6. Verify the output.