Odd/Even/Compact* Semester (year)

BINUS UNIVERSITY INTERNATIONAL

BINUS UNIVERSITY
BINUS INTERNATIONAL

Assignment Cover Letter (Individual Work)

Student Information:	*Surname*	*Given Names*	*Student ID Number*
	Hartono	Aimee	2301910322

Course Code	: COMP6510	**Course Name**	: Programming languages
Class	: L2AC -LEC	**Name of Lecturer(s)**	: JudeMartinez
Major	: Computer Science		
Title of Assignment (if any)	:		
Type of Assignment	: Project		
Submission Pattern			
Due Date	: 20-06-2020	**Submission Date**	: 20-06-2020

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I/ declare that the work contained in this assignment is my/our* own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

(Name of Student)

Aimee P. Hartono

**) Delete the inappropriate option*

Project Description

The application that I created is a *Recipe Book.* Users can start the application by loading a pre-existing XML file of the recipes or start anew with an empty recipe table. The three main features of this application are “Edit”, “Delete” and “New”. With “Edit”, users can select any recipe from the table and edit the recipe details - name, cuisine type, course and duration of preparation - and the ingredients and procedures of the dish. Users can also “Delete” existing recipe records by selecting it and clicking on the *Delete* button. The “New” button allows users to enter a new recipe into the given textfields and add the ingredients and procedures into the given textareas. Before closing the application, users can choose to save the recipe book or saveAs to a new file, and the next time the application opens it will display the most recent file loaded in the application.

My motivation starting this project was that I have always been interested in cooking, yet I would find it difficult to constantly look up the recipe online or through a book. I was then inspired to create an application that allows the addition and deletion of recipes to my own desire, and storing only the recipes that I intend to keep.

Solution Design

The classes that are involved in my project are:

- Recipe.java
- RecipeListWrapper.java
- EditRecipeController.java
- EditRecipe.fxml
- RecipePageController.java
- RecipePage.fxml
- RootLayoutController.java
- RootLayout.fxml
- Theme.css
- Main.java

Note: The data entered by the user will be persisted using XML instead of a database. This is due to the fact that the data to be saved are objects, which are not the type or relational data that are commonly used in databases. The library used for generating the XML output will be JAXB(Java Architecture for XML Binding).

Recipe.java

This class holds the constructor method, as well as getter and setter methods of the recipe objects.

```
3   import javafx.beans.property.IntegerProperty;
4   import javafx.beans.property.SimpleIntegerProperty;
5   import javafx.beans.property.SimpleStringProperty;
6   import javafx.beans.property.StringProperty;
7
8   public class Recipe {
9
10      private final StringProperty dishname;
11      private final StringProperty dishtype;
12      private final StringProperty course;
13      private final IntegerProperty duration;
14
15      private String ingredients;
16      private String procedures;
17
18      public Recipe() {
19          this( dishname: null);
20      }
21
22      public Recipe(String dishname){
```

These are the imported java classes and the main variables for the recipe objects.

The variables *dishname, dishtype, course and duration* are defined as StringProperties, since they will be observed later on.

Following that are the Recipe constructors, which uses ‘dishname’ as its main variable since it is already a unique key.

```
    public int getDuration() { return duration.get(); }

    public IntegerProperty durationProperty() { return duration; }

    public void setDuration(int duration) { this.duration.set(duration); }

    public String getDishname() { return dishname.get(); }

    public StringProperty dishnameProperty() { return dishname; }

    public void setDishname(String dishname) { this.dishname.set(dishname); }

    public String getDishtype() { return dishtype.get(); }

    public StringProperty dishtypeProperty() { return dishtype; }

    public void setDishtype(String dishtype) { this.dishtype.set(dishtype); }

    public String getCourse() { return course.get(); }

    public StringProperty courseProperty() { return course; }

    public void setCourse(String course) { this.course.set(course); }

    public String getIngredients() { return ingredients; }

    public void setIngredients(String ingredients) { this.ingredients = ingredients; }

    public String getProcedures() { return procedures; }

    public void setProcedures(String procedures) { this.procedures = procedures; }
}
```

The following are getter and setter methods for each of the mentioned variables, which will be used in the other classes.

EditRecipeController.java

This is a controller class for the EditRecipe.fxml file. Its purpose is to initiate a dialog box that allows the user to edit the recipe details, ingredients and procedures, as well as saving or cancelling the changes.

```
package recipes.view;

import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.stage.Stage;
import recipes.model.Recipe;

public class EditRecipeController {
    // to let the FXML access private classes
    @FXML
    private TextField dishnameField;
    @FXML
    private TextField dishtypeField;
    @FXML
    private TextField courseField;
    @FXML
    private TextField durationField;

    @FXML
    private TextArea ingredientsField;
    @FXML
    private TextArea proceduresField;

    private Stage dialogStage;
    private Recipe recipe;
    private boolean saveClick = false;

    @FXML
    private void initialize(){  // to initialize the controller
    }
```

These are the imported java classes, as well as references to the fxml file, where the textfields and textareas are defined. Here, the Stage dialogStage is defined and saveClick is set to false for use in another method later on.

The initialize method is created to initialize the controller.

```
    public void setDialogStage(Stage dialogStage) { this.dialogStage = dialogStage; }

    public void setRecipe(Recipe recipe){ // sets up the food to be put in the dialog, can be called from another class to set the dish to be edited
        this.recipe = recipe;

        dishnameField.setText(recipe.getDishname());
        dishtypeField.setText(recipe.getDishtype());
        courseField.setText(recipe.getCourse());
        durationField.setText(Integer.toString(recipe.getDuration()));

        ingredientsField.setText(recipe.getIngredients());
        proceduresField.setText(recipe.getProcedures());
        ingredientsField.setWrapText(true);
        proceduresField.setWrapText(true);
    }
```

The method setDialogStage sets up the dialog stage for editing the recipe. Following this method is the setRecipe method, which takes in Recipe as a variable in order to set up the recipe objects to be put in the dialog, taken from the Recipe class.

```
    public boolean isSaveClick() { return saveClick; }

    private boolean isValid() { // to validate the user input for each field
        String error = "";

        if (dishnameField.getText() == null || dishnameField.getText().length() == 0) {
            error += "Dish Name invalid :( \n";
        }
        if (dishtypeField.getText() == null || dishtypeField.getText().length() == 0) {
            error += "Cuisine Type invalid :( \n";
        }
        if (courseField.getText() == null || courseField.getText().length() == 0) {
            error += "Course invalid :( \n";
        }
        if (durationField.getText() == null || durationField.getText().length() == 0) {
            error += "Duration invalid :( \n";
        } else {
            // try to parse the duration into an int
            try {
                Integer.parseInt(durationField.getText());
            } catch (NumberFormatException e) {
                error += "Duration must be an integer :( \n";
            }
        }
```

The isSaveClick method returns saveClick, which was previously set as *false*. This will be used for when the user decides to click on the save button after editing the recipe details. The isValid method will check whether or not the user inputs are valid and will append it to the error string if it is invalid(since the durationField only accepts integers, it will parse the input into an integer to check whether or not it is a valid integer).

```
        if (error.length() == 0) {
            return true;    // will return true if the error message is empty
        } else {    // error message will be shown
            Alert alert = new Alert(Alert.AlertType.ERROR);
            alert.initOwner(dialogStage);
            alert.setTitle("Is this some advanced language that I don't understand?");
            alert.setHeaderText("NOT SO FAST! I get that you're excited, but please enter the fields correctly.");
            alert.setContentText(error);
            alert.showAndWait();

            return false;
        }
    }
```

The next part of the code will then check if there is an error and will show an alert message of alert type *ERROR* that will notify the user of the invalidity in their input.

```
    public void saveButton() {  // for when the user wants to save the recipe
        if (isValid()) {
            recipe.setDishname(dishnameField.getText());
            recipe.setDishtype(dishtypeField.getText());
            recipe.setCourse(courseField.getText());
            recipe.setDuration(Integer.parseInt(durationField.getText()));

            recipe.setIngredients(ingredientsField.getText());
            recipe.setProcedures(proceduresField.getText());

            saveClick = true;
            dialogStage.close();
        }
    }

    public void cancelButton() {    // for when the user decides to cancel
        dialogStage.close();
    }
```

The saveButton method will set the recipe details, as well as ingredients and procedures, after checking that the inputs are valid. saveClick will then be set to *true* to indicate that saving is permitted, and the dialog stage will close automatically.
The cancelButton method will close the dialog stage without saving the changes.

RecipePageController.java

This is a controller class for the RecipePage.fxml file. It is the main page, in which its purpose is to display the tableview that contains the recipe details, which can each be observed to show the corresponding ingredients and procedures. It also holds the buttonbar that will allow the user to *edit* or *delete* a chosen recipe, or add a *new* one.

```java
package recipes.view;

import javafx.collections.transformation.FilteredList;
import javafx.collections.transformation.SortedList;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import recipes.Main;
import recipes.model.*;

public class RecipePageController {
    // to let the FXML access private classes
    @FXML
    private TextField searchDish;
    @FXML
    private TableView<Recipe> dishlist;
    @FXML
    private TableColumn<Recipe, Integer> durationColumn;
    @FXML
    private TableColumn<Recipe, String> dishnameColumn;
    @FXML
    private TableColumn<Recipe, String> dishtypeColumn;
    @FXML
    private TableColumn<Recipe, String> courseColumn;

    @FXML
    private Label dishnameLabel;
    @FXML
    private TextArea ingredientsArea;
    @FXML
    private TextArea proceduresArea;

    // referencing the main application
    private recipes.Main Main;
```

These are the imported java classes, as well as references to the fxml file for each of the variables that will be displayed in the main page.

The columns *durationColumn, dishnameColumn, dishtypeColumn and courseColumn* are contained in the tableview *dishlist*, and the textfield *searchDish* will be used for filtering the table data. There will also be spaces for the ingredients and procedures to be displayed(*ingredientsArea* and *proceduresArea* respectively).

```java
    public RecipePageController() {
    }

    // for displaying the ingredients and procedures pages
    public void showRecipe(Recipe recipe) {
        if (recipe != null) {
            //fill the text with ingredients for that dish
            ingredientsArea.setText(recipe.getIngredients());
            proceduresArea.setText(recipe.getProcedures());
            dishnameLabel.setText(recipe.getDishname());

        } else {
            //field will be empty if the food is null
            ingredientsArea.setText("");
            proceduresArea.setText("");
            dishnameLabel.setText("no recipe selected");
        }
    }
```

The showRecipe method displays the corresponding ingredients and procedures - in *ingredientsArea* and *proceduresArea* - as well as the name of the recipe clicked by the user.

```java
    public void setMain(Main main) {
        this.Main = main;

        // Add observable list data to the table
        dishlist.setItems(main.getRecipeData());
```

The setMain method sets up the main data.

Firstly, it adds the observable list that holds the recipe objects - located in Main.java - to the *dishlist* tableview.

```
        //filter
        // 1. wrap the ObservableList in a FilteredList, display all the data initially
        FilteredList<Recipe> filteredData = new FilteredList<>(main.getRecipeData(), p -> true);
        //this indicates that the first predicate is always true
```

This is the first part of setting up the table filter. It will wrap the recipeData observable list from Main.java in a filtered list.

```
    // 2. set the predicate in the filter for whenever the filter changes
    searchDish.textProperty().addListener((observable, oldValue, newValue) -> { // ChangeListener is added to the filter text field
        filteredData.setPredicate(recipe -> {                                    // the predicate of the FilteredList is updated whenever
                                                                                 // the user changes the text
            if (newValue == null || newValue.isEmpty()) {
                return true;    // display all recipes in the table if the filter is empty
            }

            // compare the dish details with the keyword entered in the filter
            String lowerCaseFilter = newValue.toLowerCase();

            if (recipe.getDishname().toLowerCase().contains(lowerCaseFilter)) {
                return true;    // indicates that the filter matches the dish name
                // repeat for the rest of the details
            } else if (recipe.getDishtype().toLowerCase().contains(lowerCaseFilter)) {
                return true;
            } else if (recipe.getCourse().toLowerCase().contains(lowerCaseFilter)) {
                return true;
            } else if (recipe.getDishname().toLowerCase().contains(lowerCaseFilter)) {
                return true;
            }
            return false;   // for when the filter keyword does not match any entry
        });
    });
```

The second part will get the relevant recipe details according to the predicate that the user enters into the textfield. Initially, it will display all the contents while the user has not entered anything into the search textfield. As the user starts to enter the letters, the predicate of the filteredlist will keep updating and will be compared with sections of each recipe detail. If the filter matches, it will return *true* and *false* otherwise.

```
        // 3. wrap the FilteredList in a SortedList
        SortedList<Recipe> sortedData = new SortedList<>(filteredData);
                // since FilteredList cannot be modified, it cannot be sorted.
                // therefore it needs to be wrapped in a sorted list

        // 4. bind the SortedList comparator to the TableView comparator
        sortedData.comparatorProperty().bind(dishlist.comparatorProperty());

        // 5. add the sorted and filtered data to the table
        dishlist.setItems(sortedData);
    }
```

In the next parts, the filteredlist is wrapped into a sortedlist in order to allow changes to the list. Its comparator will then be binded to that of the tableview, in which its contents will be updated to what is filtered.

```
100     @FXML // delete function
101     private void deleteDish(){
102         int deleteIndex = dishlist.getSelectionModel().getSelectedIndex();
103 
104         if (deleteIndex >= 0) {
105             Main.getRecipeData().remove(deleteIndex);
106         } else {    // in case the user selects an empty dish
107             Alert alert = new Alert(Alert.AlertType.WARNING);   // a pop-up alert will appear
108             alert.initOwner(Main.getPrimaryStage());
109             alert.setTitle("Empty Selection");
110             alert.setHeaderText("Were you about to delete... Nothing? ");
111             alert.setContentText("At least pick something from the list");
112             alert.showAndWait();
113         }
114     }
```

The deleteDish method controls the *delete* button in the main page to allow users to delete any recipe of their choosing. If no recipes are selected, a *WARNING* alert will be shown.

```
116     @FXML
117     private void newDish() {    // for when the user wants to save a new recipe
118         Recipe newRecipe = new Recipe();
119         boolean saveClick = Main.show_EditRecipe(newRecipe);    // the Edit Recipe dialog will be shown
120         if (saveClick) {
121             Main.getRecipeData().add(newRecipe);
122         }
123     }
```

The newDish method controls the *new* button, which references Main.java to show the editRecipe dialog and adds the new recipe to the recipeData observablelist.

```
125     @FXML
126     private void editDish() {   // for when the user wants to edit an existing recipe
127         Recipe chosenRecipe = dishlist.getSelectionModel().getSelectedItem();
128         if (chosenRecipe != null) {
129             boolean saveClick = Main.show_EditRecipe(chosenRecipe);   // the Edit Recipe dialog will be shown
130             if (saveClick) {
131                 showRecipe(chosenRecipe);
132             }
133         } else {    // if no dish have been selected, an alert will pop up
134             Alert alert = new Alert(Alert.AlertType.WARNING);
135             alert.initOwner(Main.getPrimaryStage());
136             alert.setTitle("No Selection");
137             alert.setHeaderText("I see no dish have been brought forth.");
138             alert.setContentText("Please choose a dish from the list. It's really not that hard.");
139             alert.showAndWait();
140         }
141     }
```

The editDish method controls the *edit* button, which will take the recipe that was clicked or chosen by the user and open the editRecipe dialog - it still contains the original details of the recipe. If no recipes are selected, a *WARNING* alert will be shown.

```
    @FXML
    public void initialize() {  //to initialize the food table with the columns
        // the following code ensures that the fields in the Food objects are used in their respective columns
        durationColumn.setCellValueFactory(cellData -> cellData.getValue().durationProperty().asObject());
        dishnameColumn.setCellValueFactory(cellData -> cellData.getValue().dishnameProperty());
        dishtypeColumn.setCellValueFactory(cellData -> cellData.getValue().dishtypeProperty());
        courseColumn.setCellValueFactory(cellData -> cellData.getValue().courseProperty());

        // set the ingredients and procedures Text Areas to be non-editable
        ingredientsArea.setEditable(false);
        ingredientsArea.setWrapText(true);
        proceduresArea.setEditable(false);
        proceduresArea.setWrapText(true);

        // empty the field for the details of the dish
        showRecipe(null);

        //detect the selection changes and display the ingredients and procedures once selected
        dishlist.getSelectionModel().selectedItemProperty().addListener(
                (observable, oldValue, newValue) -> showRecipe(newValue));
    }


}
```

The last method in this class is the initialize method, which initializes the *dishlist* tableview with the columns. The fields in the recipe objects are placed in their respective cells within the columns and *ingredientsArea* and *proceduresArea* are set so that their fields are non-editable in the main page and their texts wrap automatically.
The main page will not display any recipe upon first opening the main page, until it detects a selection from the tableview.

RecipeListWrapper.java

This class holds the list of recipes that will be saved in the XML.

```
package recipes.model;

import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "recipes")   // to define the root element name
public class RecipeListWrapper {    // helper class to wrap the list of recipes
    private List<Recipe> recipes;

    @XmlElement(name = "recipe")    // this will be the optional name of the element
    public List<Recipe> getRecipes() { return recipes; }

    public void setRecipes(List<Recipe> recipes) { this.recipes = recipes; }
}
```

The XML root element name is defined as *recipes*. The RecipeListWrapper class is a helper class that will wrap the list of recipes, also defined as *recipes.* The XML element itself is defined as *recipe*, and the following code are getter and setter methods to set and return the recipes.

Main.java

This is the main class of the application, which holds the primary stage of the application and ensures each dialog are loaded.

```
3   import javafx.application.Application;
4   import javafx.collections.FXCollections;
5   import javafx.collections.ObservableList;
6   import javafx.fxml.FXMLLoader;
7   import javafx.scene.Scene;
8   import javafx.scene.control.Alert;
9   import javafx.scene.image.Image;
10  import javafx.stage.Modality;
11  import javafx.stage.Stage;
12
13  import javafx.scene.layout.AnchorPane;
14  import javafx.scene.layout.BorderPane;
15  import recipes.model.Recipe;
16  import recipes.model.RecipeListWrapper;
17  import recipes.view.EditRecipeController;
18  import recipes.view.RecipePageController;
19  import recipes.view.RootLayoutController;
20
21  import javax.xml.bind.JAXBContext;
22  import javax.xml.bind.Marshaller;
23  import javax.xml.bind.Unmarshaller;
24  import java.io.File;
25  import java.io.IOException;
26  import java.util.prefs.Preferences;
```

These are the imported java classes for this class.

```
28  public class Main extends Application {
29
30      private Stage primaryStage;
31      private BorderPane root_layout;
32
33      private ObservableList<Recipe> dishData = FXCollections.observableArrayList();
34      public ObservableList<Recipe> getRecipeData() { return dishData; }
37
38      public Main() {
39      }
```

This is where the primary stage is defined, as well as the root layout. The observablelist *dishData* is defined here, as well as the method getRecipeData, which returns *dishData* for other classes. This is also where the Main class is constructed.

```
47      @Override
48      public void start(Stage primaryStage) {
49          this.primaryStage = primaryStage;
50          this.primaryStage.setTitle("Recipe App");
51
52          // sets the icon of the application
53          this.primaryStage.getIcons().add(new Image( s: "file: icon/chef.png"));
54
55          rootLayout_init();
56          show_recipePage();
57
58      }
```

The start method sets the primary stage and its title, as well as the window icon. Following that, the root layout is initialized and the main page is shown.

```
60      // initializing the root layout and loading the last opened recipe file
61      public void rootLayout_init() {
62          try {
63              // loading the root layout from the FXML file
64              FXMLLoader loader = new FXMLLoader();
65              loader.setLocation(Main.class.getResource( name: "view/RootLayout.fxml"));
66              root_layout = (BorderPane) loader.load();
67
68              // showing the root layout scene
69              Scene scene = new Scene(root_layout);
70              primaryStage.setScene(scene);
71
72              // allowing access for the root layout controller
73              RootLayoutController controller = loader.getController();
74              controller.setMain(this);
75
76              primaryStage.show();
```

The rootLayout_init method will try to load the RootLayout.fxml file and show the root layout scene on the primary stage. It then allows access for the root layout controller to take action.

```
78          } catch (IOException e) {
79              e.printStackTrace();
80          }
81
82          // load the last opened file
83          File file = getRecipePath();
84          if (file != null) {
85              loadRecipeDataFromFile(file);
86          }
87
88      }
```

Otherwise, it will catch the exception.
Following that, it will load the last-opened recipe file automatically.

```
90      public void show_recipePage() {
91          try {
92              // loading the recipe page from the FXML file
93              FXMLLoader loader = new FXMLLoader();
94              loader.setLocation(Main.class.getResource( name: "view/RecipePage.fxml"));
95              AnchorPane RecipePage = (AnchorPane) loader.load();
96
97              // setting the recipe page in the center of the root layout
98              root_layout.setCenter(RecipePage);
99
100             // allowing the controller access to the main application
101             RecipePageController controller = loader.getController();
102             controller.setMain(this);
103
104         } catch (IOException e) {
105             e.printStackTrace();
106         }
107     }
```

The show_recipePage method will try to load the RecipePage.fxml file and set it in the center of the root layout. It will then allow access for the recipe page controller to take action. Otherwise, it will catch the exception.

```
104 public boolean show_EditRecipe(Recipe recipe) { // load and display the Edit Recipe dialog in the main application
105     try{
106         // a new pop-up stage will be created
107         FXMLLoader loader = new FXMLLoader();
108         loader.setLocation(Main.class.getResource( name: "view/EditRecipe.fxml"));
109         AnchorPane page = loader.load();
110
111         //the dialog stage
112         Stage dialogStage = new Stage();
113         dialogStage.setTitle("Edit Recipe");
114         dialogStage.initModality(Modality.WINDOW_MODAL);
115         dialogStage.initOwner(primaryStage);
116         Scene scene = new Scene(page);
117         dialogStage.setScene(scene);
```

The show_EditRecipe method will try to load and display the EditRecipe.fxml file. It will set a new stage *dialogStage*, which will be a modal window that blocks events from being delivered to the owner window, *primaryStage*.

```
            //set the food into the controller
            EditRecipeController controller = loader.getController();
            controller.setDialogStage(dialogStage);
            controller.setRecipe(recipe);

            // dialog will be shown until the user closes it
            dialogStage.showAndWait();

            return controller.isSaveClick();

        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
```

Next, it will set the recipes into the controller and will set *dialogStage* so that it will be shown until the user closes.
Otherwise, it will catch the exception.

```
    public Stage getPrimaryStage() {
        return primaryStage;
    }
```

The getPrimaryStage method returns *primaryStage.*

```
public File getRecipePath() {   // returns the last preference of the app(last file opened)
    Preferences preferences = Preferences.userNodeForPackage(Main.class);
    String path = preferences.get( key: "path",  def: null);
    if (path != null) {
        return new File(path);
    } else {
        return null;    // null is returned if the preferences cannot be found
    }
}
```

The method getRecipePath returns the recipe file that was last opened. If it does not find any, it will return null.

```
156     public void setRecipePath(File file) {
157         // sets the file path of the current file(will be persisted in the OS specific registry
158         Preferences preferences = Preferences.userNodeForPackage(Main.class);
159         if (file != null) {
160             preferences.put("path", file.getPath());
161 
162             // stage title will be updated
163             primaryStage.setTitle("Recipe App - " + file.getName());
164         } else {
165             preferences.remove( key: "path");
166 
167             // stage title will be updated
168             primaryStage.setTitle("Recipe App");
169         }
170     }
```

The setRecipePath method will set the file path of the currently opened file and update the stage title to contain the file path. If none is found, stage title will stay as the default stage title.

```
172     public void loadRecipeDataFromFile(File file) {
173         //loads the dish data from the specified file and replace the current one
174         try {
175             JAXBContext context = JAXBContext.newInstance(RecipeListWrapper.class);
176             Unmarshaller um = context.createUnmarshaller();
177 
178             // XML will be read from the file and unmarshalling occurs
179             RecipeListWrapper wrapper = (RecipeListWrapper) um.unmarshal(file);
180 
181             dishData.clear();
182             dishData.addAll(wrapper.getRecipes());
183 
184             // saving the file to the path registry
185             setRecipePath(file);
186         } catch (Exception e) { // for any exception
187             Alert alert = new Alert(Alert.AlertType.ERROR);
188             alert.setTitle("Error");
189             alert.setHeaderText("Houston, we have a problem.");
190             alert.setContentText("hmm... it seems we can't load from file:\n" + file.getPath());
191             alert.showAndWait();
192         }
193     }
```

The loadRecipeDataFromFile method will try to load the specified file into the application and replace the current one. Using JAXB, the file will be unmarshalled into XML. After that, the observablelist *dishData* will be cleared and replaced with the contents of *wrapper.* The file will then be saved to the current path registry.
Otherwise, it will catch the exception and show an *ERROR* alert that notifies how the file is unable to be saved at that path.

```
197     public void saveRecipeDataToFile (File file) {  // the recipe data will be saved to the given file
198         try {
199             JAXBContext context = JAXBContext.newInstance(RecipeListWrapper.class);
200             Marshaller m = context.createMarshaller();
201             m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
202 
203             // dish data is wrapped
204             RecipeListWrapper wrapper = new RecipeListWrapper();
205             wrapper.setRecipes(dishData);
206 
207             // marshalling and saving the XML to the file
208             m.marshal(wrapper, file);
209 
210             // file path is saved to the registry
211             setRecipePath(file);
212         } catch (Exception e) { // for every exception
213             Alert alert = new Alert(Alert.AlertType.ERROR);
214             alert.setTitle("Error");
215             alert.setHeaderText("Uh-oh, we've found ourselves an error.");
216             alert.setContentText("hmm... it seems we can't save to file:\n" + file.getPath());
217             alert.showAndWait();
218         }
219     }
```

The saveRecipeDataToFile method allows the user to save the recipe data to the given file. Once again with JAXB, this time it will create a marshaller. After the recipes are set in the observablelist *dishData,* the XML will be marshalled into Java objects and the new file will be saved to the registry.
Otherwise, it will catch an exception and notify the user with an *ERROR* alert that the file cannot be saved to the path.

```
221     public static void main(String[] args) { launch(args); }
224 }
```

This method will launch the application.

RootLayoutController.java

This is a controller class for the RootLayout.fxml file, which acts as the border of the main page and contains a menubar for saving and editing the file.

```
3   package recipes.view;
4 
5   import javafx.fxml.FXML;
6   import javafx.scene.control.Alert;
7   import javafx.scene.control.Alert.AlertType;
8   import javafx.stage.FileChooser;
9   import recipes.Main;
10 
11  import java.io.File;
```

These are the imported java classes for this class.

```java
public class RootLayoutController {
    // reference to the main application
    private Main main;

    public void setMain(Main main) { this.main = main; }

    @FXML
    public void selectNew() {   // creates a new recipe book
        main.getRecipeData().clear();
        main.setRecipePath(null);
    }
```

The following code references Main.java.

The selectNew method creates an entirely new recipe book by clearing the current data and file path.

```java
    @FXML
    private void selectOpen() {  // allows user to load an existing recipe book
        FileChooser fileChooser = new FileChooser();

        // extension filter is set so that only files ending with .xml are displayed
        FileChooser.ExtensionFilter extensionFilter = new FileChooser.ExtensionFilter( s: "XML files (*.xml)", ...strings: "*xml");
        fileChooser.getExtensionFilters().add(extensionFilter);

        // show dialog for when Open is selected
        File file = fileChooser.showOpenDialog(main.getPrimaryStage());

        if (file != null) {
            main.loadRecipeDataFromFile(file);
        }   // null is returned if the user closes the dialog without choosing a file
    }
```

The selectOpen method allows the user to choose an existing file to load into the recipe application. It makes use of a filechooser to filter the extensions of the file(it will be looking for xml files), then showing the Open dialog.

```java
    @FXML
    private void selectSave() { // saves the file to the currently opened one,
                                // will show the "Save As" dialog if no file is opened
        File recipeFile = main.getRecipePath();
        if (recipeFile != null) {
            main.saveRecipeDataToFile(recipeFile);
        } else {
            selectSaveAs();
        }
    }
```

The selectSave method allows users to save the current file to the current directory, but will open the selectSaveAs method instead if it is a new file.

```
43      @FXML
44      private void selectSaveAs() {   // allows user to select a file to save to
45          FileChooser fileChooser = new FileChooser();
46
47          // extension filter is set
48          FileChooser.ExtensionFilter extensionFilter = new FileChooser.ExtensionFilter( s: "XML files (*.xml)",  ...strings: "*.xml");
49          fileChooser.getExtensionFilters().add(extensionFilter);
50
51          // save file dialog is shown
52          File file = fileChooser.showSaveDialog(main.getPrimaryStage());
53
54          if (file != null) { // null is returned if the user closes the dialog without choosing a file
55              // ensure that the file extension is correct
56              if (!file.getPath().endsWith(".xml")) {
57                  file = new File( pathname: file.getPath() + ".xml");
58              }
59              main.saveRecipeDataToFile(file);
60          }
61      }
```

The selectSaveAs method also uses a filechooser to filter the extension of the files, before showing the Save dialog. It will ensure that the file extension is correct(has to be xml) before saving the file to the chosen directory.

```
82      @FXML
83      private void selectClose() { System.exit( status: 0); }
86
87  }
```

If the user chooses to close the application, the selectClose method will exit the application.

Resources:

- Jakob, Marco “JavaFX Tutorial” *code.makery,* 12th March 2015, https://code.makery.ch/library

Modules used:

- javafx-sdk-14.0.1
- jaxb-api-2.3.1\jaxb-ri