



Assignment Cover Letter (Individual/Group* Work)

Student Information:

<i>Surname</i>	<i>Given Names</i>	<i>Student ID Number</i>
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/we* understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I/we* 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:

A handwritten signature in black ink, appearing to be "Aimee P. Hartono".

(Name of Student)

Aimee P. Hartono

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.

```

37 public int getDuration() { return duration.get(); }
40
41 public IntegerProperty durationProperty() { return duration; }
44
45 public void setDuration(int duration) { this.duration.set(duration); }
48
49 public String getDishname() { return dishname.get(); }
52
53 public StringProperty dishnameProperty() { return dishname; }
56
57 public void setDishname(String dishname) { this.dishname.set(dishname); }
60
61 public String getDishtype() { return dishtype.get(); }
64
65 public StringProperty dishtypeProperty() { return dishtype; }
68
69 public void setDishtype(String dishtype) { this.dishtype.set(dishtype); }
72
73 public String getCourse() { return course.get(); }
76
77 public StringProperty courseProperty() { return course; }
80
81 public void setCourse(String course) { this.course.set(course); }
84
85 public String getIngredients() { return ingredients; }
88
89 public void setIngredients(String ingredients) { this.ingredients = ingredients; }
92
93 public String getProcedures() { return procedures; }
96
97 public void setProcedures(String procedures) { this.procedures = procedures; }
100
}

```

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.

```

1 package recipes.view;
2
3 import javafx.fxml.FXML;
4 import javafx.scene.control.*;
5 import javafx.stage.Stage;
6 import recipes.model.Recipe;
7
8 public class EditRecipeController {
9     // to let the FXML access private classes
10
11     @FXML
12     private TextField dishnameField;
13
14     @FXML
15     private TextField dishtypeField;
16
17     @FXML
18     private TextField courseField;
19
20     @FXML
21     private TextField durationField;
22
23     @FXML
24     private TextArea ingredientsField;
25
26     @FXML
27     private TextArea proceduresField;
28
29     private Stage dialogStage;
30     private Recipe recipe;
31     private boolean saveClick = false;
32
33     @FXML
34     private void initialize(){ // to initialize the controller
35     }
36 }

```

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.

```

32 public void setDialogStage(Stage dialogStage) { this.dialogStage = dialogStage; }
35
36 @
37 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
38     this.recipe = recipe;
39
40     dishnameField.setText(recipe.getDishname());
41     dishtypeField.setText(recipe.getDishtype());
42     courseField.setText(recipe.getCourse());
43     durationField.setText(Integer.toString(recipe.getDuration()));
44
45     ingredientsField.setText(recipe.getIngredients());
46     proceduresField.setText(recipe.getProcedures());
47     ingredientsField.setWrapText(true);
48     proceduresField.setWrapText(true);
49 }

```

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.

```

50 public boolean isSaveClick() { return saveClick; }
53
54 private boolean isValid() { // to validate the user input for each field
55     String error = "";
56
57     if (dishnameField.getText() == null || dishnameField.getText().length() == 0) {
58         error += "Dish Name invalid :( \n";
59     }
60     if (dishtypeField.getText() == null || dishtypeField.getText().length() == 0) {
61         error += "Cuisine Type invalid :( \n";
62     }
63     if (courseField.getText() == null || courseField.getText().length() == 0) {
64         error += "Course invalid :( \n";
65     }
66     if (durationField.getText() == null || durationField.getText().length() == 0) {
67         error += "Duration invalid :( \n";
68     } else {
69         // try to parse the duration into an int
70         try {
71             Integer.parseInt(durationField.getText());
72         } catch (NumberFormatException e) {
73             error += "Duration must be an integer :( \n";
74         }
75     }
76 }

```

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).

```

77     if (error.length() == 0) {
78         return true; // will return true if the error message is empty
79     } else { // error message will be shown
80         Alert alert = new Alert(Alert.AlertType.ERROR);
81         alert.initOwner(dialogStage);
82         alert.setTitle("Is this some advanced language that I don't understand?");
83         alert.setHeaderText("NOT SO FAST! I get that you're excited, but please enter the fields correctly.");
84         alert.setContentText(error);
85         alert.showAndWait();
86
87         return false;
88     }
89 }

```

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.

```

91     public void saveButton() { // for when the user wants to save the recipe
92         if (isValid()) {
93             recipe.setDishname(dishnameField.getText());
94             recipe.setDishtype(dishtypeField.getText());
95             recipe.setCourse(courseField.getText());
96             recipe.setDuration(Integer.parseInt(durationField.getText()));
97
98             recipe.setIngredients(ingredientsField.getText());
99             recipe.setProcedures(proceduresField.getText());
100
101             saveClick = true;
102             dialogStage.close();
103         }
104     }
105
106     public void cancelButton() { // for when the user decides to cancel
107         dialogStage.close();
108     }

```

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.

```
1 package recipes.view;
2
3 import javafx.collections.transformation.FilteredList;
4 import javafx.collections.transformation.SortedList;
5 import javafx.fxml.FXML;
6 import javafx.scene.control.*;
7 import recipes.Main;
8 import recipes.model.*;
9
10 public class RecipePageController {
11     // to let the FXML access private classes
12     @FXML
13     private TextField searchDish;
14     @FXML
15     private TableView<Recipe> dishlist;
16     @FXML
17     private TableColumn<Recipe, Integer> durationColumn;
18     @FXML
19     private TableColumn<Recipe, String> dishnameColumn;
20     @FXML
21     private TableColumn<Recipe, String> dishtypeColumn;
22     @FXML
23     private TableColumn<Recipe, String> courseColumn;
24
25     @FXML
26     private Label dishnameLabel;
27     @FXML
28     private TextArea ingredientsArea;
29     @FXML
30     private TextArea proceduresArea;
31
32     // referencing the main application
33     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).

```
35 public RecipePageController() {
36 }
37
38 // for displaying the ingredients and procedures pages
39 public void showRecipe(Recipe recipe) {
40     if (recipe != null) {
41         //fill the text with ingredients for that dish
42         ingredientsArea.setText(recipe.getIngredients());
43         proceduresArea.setText(recipe.getProcedures());
44         dishnameLabel.setText(recipe.getDishname());
45     } else {
46         //field will be empty if the food is null
47         ingredientsArea.setText("");
48         proceduresArea.setText("");
49         dishnameLabel.setText("no recipe selected");
50     }
51 }
52 }
```

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.

```
54 @ public void setMain(Main main) {
55     this.Main = main;
56
57     // Add observable list data to the table
58     dishlist.setItems(main.getRecipeData());
59 }
```

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.

```
60 //filter
61 // 1. wrap the ObservableList in a FilteredList, display all the data initially
62 FilteredList<Recipe> filteredData = new FilteredList<>(main.getRecipeData(), p -> true);
63 //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.

```
64 // 2. set the predicate in the filter for whenever the filter changes
65 searchDish.textProperty().addListener((observable, oldValue, newValue) -> { // ChangeListener is added to the filter text field
66     filteredData.setPredicate(recipe -> { // the predicate of the FilteredList is updated whenever
67                                         // the user changes the text
68         if (newValue == null || newValue.isEmpty()) {
69             return true; // display all recipes in the table if the filter is empty
70         }
71
72         // compare the dish details with the keyword entered in the filter
73         String lowerCaseFilter = newValue.toLowerCase();
74
75         if (recipe.getDishname().toLowerCase().contains(lowerCaseFilter)) {
76             return true; // indicates that the filter matches the dish name
77             // repeat for the rest of the details
78         } else if (recipe.getDishtype().toLowerCase().contains(lowerCaseFilter)) {
79             return true;
80         } else if (recipe.getCourse().toLowerCase().contains(lowerCaseFilter)) {
81             return true;
82         } else if (recipe.getDishname().toLowerCase().contains(lowerCaseFilter)) {
83             return true;
84         }
85         return false; // for when the filter keyword does not match any entry
86     });
87 });
```

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.

```
89 // 3. wrap the FilteredList in a SortedList
90 SortedList<Recipe> sortedData = new SortedList<>(filteredData);
91 // since FilteredList cannot be modified, it cannot be sorted.
92 // therefore it needs to be wrapped in a sorted list
93
94 // 4. bind the SortedList comparator to the TableView comparator
95 sortedData.comparatorProperty().bind(dishlist.comparatorProperty());
96
97 // 5. add the sorted and filtered data to the table
98 dishlist.setItems(sortedData);
99 }
```

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.

```

143     @FXML
144     public void initialize() { //to initialize the food table with the columns
145         // the following code ensures that the fields in the Food objects are used in their respective columns
146         durationColumn.setCellValueFactory(cellData -> cellData.getValue().durationProperty().asObject());
147         dishnameColumn.setCellValueFactory(cellData -> cellData.getValue().dishnameProperty());
148         dishtypeColumn.setCellValueFactory(cellData -> cellData.getValue().dishtypeProperty());
149         courseColumn.setCellValueFactory(cellData -> cellData.getValue().courseProperty());
150
151         // set the ingredients and procedures Text Areas to be non-editable
152         ingredientsArea.setEditable(false);
153         ingredientsArea.setWrapText(true);
154         proceduresArea.setEditable(false);
155         proceduresArea.setWrapText(true);
156
157         // empty the field for the details of the dish
158         showRecipe(null);
159
160         //detect the selection changes and display the ingredients and procedures once selected
161         dishlist.getSelectionModel().selectedItemProperty().addListener(
162             (observable, oldValue, newValue) -> showRecipe(newValue));
163     }
164
165
166 }

```

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.

```

2     package recipes.model;
3
4     import java.util.List;
5
6     import javax.xml.bind.annotation.XmlElement;
7     import javax.xml.bind.annotation.XmlRootElement;
8
9     @XmlRootElement(name = "recipes") // to define the root element name
10    public class RecipeListWrapper { // helper class to wrap the list of recipes
11        private List<Recipe> recipes;
12
13        @XmlElement(name = "recipe") // this will be the optional name of the element
14        public List<Recipe> getRecipes() { return recipes; }
15
16
17
18        public void setRecipes(List<Recipe> recipes) { this.recipes = recipes; }
19
20
21    }

```

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; }
35
36
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("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("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.

```

77
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);
118
119         }

```

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*.


```

121
122     //set the food into the controller
123     EditRecipeController controller = loader.getController();
124     controller.setDialogStage(dialogStage);
125     controller.setRecipe(recipe);
126
127     // dialog will be shown until the user closes it
128     dialogStage.showAndWait();
129
130     return controller.isSaveClick();
131
132 } catch (IOException e) {
133     e.printStackTrace();
134     return false;
135 }
136 }

```

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.

```

138     public Stage getPrimaryStage() {
139         return primaryStage;
140     }
141

```

The `getPrimaryStage` method returns *primaryStage*.

```

146     public File getRecipePath() { // returns the last preference of the app(last file opened)
147         Preferences preferences = Preferences.userNodeForPackage(Main.class);
148         String path = preferences.get( key: "path", def: null);
149         if (path != null) {
150             return new File(path);
151         } else {
152             return null; // null is returned if the preferences cannot be found
153         }
154     }

```

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.

```

13     public class RootLayoutController {
14         // reference to the main application
15         private Main main;
16
17         public void setMain(Main main) { this.main = main; }
18
19
20
21         @FXML
22         public void selectNew() { // creates a new recipe book
23             main.getRecipeData().clear();
24             main.setRecipePath(null);
25         }

```

The following code references Main.java.

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

```

27     @FXML
28     private void selectOpen() { // allows user to load an existing recipe book
29         FileChooser fileChooser = new FileChooser();
30
31         // extension filter is set so that only files ending with .xml are displayed
32         FileChooser.ExtensionFilter extensionFilter = new FileChooser.ExtensionFilter("XML files (*.xml)", ...strings: "*.xml");
33         fileChooser.getExtensionFilters().add(extensionFilter);
34
35         // show dialog for when Open is selected
36         File file = fileChooser.showOpenDialog(main.getPrimaryStage());
37
38         if (file != null) {
39             main.loadRecipeDataFromFile(file);
40         } // null is returned if the user closes the dialog without choosing a file
41     }

```

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.

```

43     @FXML
44     private void selectSave() { // saves the file to the currently opened one,
45                             // will show the "Save As" dialog if no file is opened
46         File recipeFile = main.getRecipePath();
47         if (recipeFile != null) {
48             main.saveRecipeDataToFile(recipeFile);
49         } else {
50             selectSaveAs();
51         }
52     }

```

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("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

RECIPE BOOK

aimée p. hartono 2301910322

