




Application Program Development

Segment : Preparing Controller Classes for GUI

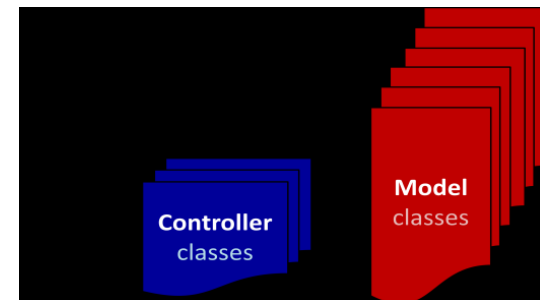
Mahboob Ali

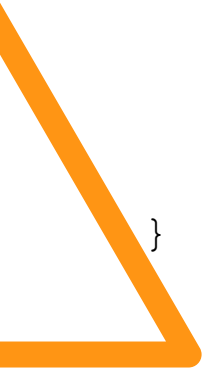



Outcomes

- Preparing Controller classes
 - Proper design need and implementation
- 

- The final piece of our application is the controller.
- The controller is responsible for taking in user input and making changes to the model accordingly.
- These changes can then be refreshed with a simple call to **update()** in the view class.
- The controller will be our **Application** class.
- It will tie together the model and the view.
- In addition, the controller is where we "hook up" our event handlers to handle the user interaction.
- It will handle all user input and then change the model accordingly ... updating the view afterwards.
- To begin, we can define the class such that it creates a new view and a new model.
- Here is the basic structure ... we will be adding the event handlers one by one.





```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class GroceryListApp extends Application {
    private ItemList model; // model to which this view is attached
    private GroceryListView view; // view that shows the state of the model
    public void start(Stage primaryStage) {
        // Create the model and view
        model = new ItemList(); // Start with an empty list
        view = new GroceryListView(model);
        // Add the event handlers
        // ... coming in next slides ...
        primaryStage.setTitle("My Grocery List");
        primaryStage.setResizable(true);
        primaryStage.setScene(new Scene(view, 285, 205));
        primaryStage.show();
    }
    public static void main(String[] args) { launch(args); }
```

Add Items

- To add an item to the Grocery List, we will require the user to type the item into the text field and then press the **Add** button.
- Let us write an event handler for when the **Add** button is pressed.
 - It will need to get the contents of the text field and then insert that string as a new item in the model.
 - Then the view should be updated. So we should add this method to the controller (i.e., the main application):

```
// The Add Button event handler  
private void handleAddButtonPress () {  
    model.add(view.getNewItemField().getText());  
    view.update();  
}
```

- Notice that this method is **private**, since no external classes should be calling it. The code does two main things that ALL of your event handlers should do:

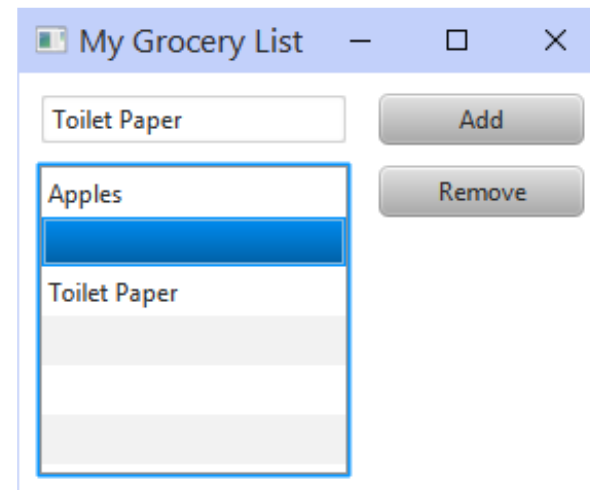
- Change the Model
- Update the View



- Now, we need to "plug it in" to the **Button**.
- We will need to add an `EventHandler<ActionEvent>` to the button.

```
//code for controller's constructor
view.getAddButton().setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent actionEvent) {
        handleAddButtonPress();
    }
});
```

- Notice how the **Button** is accessed from the view. The code hooks up to the event handler method that we just wrote. The code should now work and allow new items to be added to the list.
- However, a slight problem occurs when the user does not have anything typed into the text field and then presses the **Add** button.
 - A "blank" item is added to the list. This is not pleasant.



- We can ensure that no blank items are added by altering our code a little:

```
private void handleAddButtonPress () {  
    String text = view.getNewItemField().getText().trim();  
    if (text.length() > 0) {  
        model.add(text);  
        view.update();  
    }  
}
```

- You may have noticed the **trim()** method added here.
- This is a **String** method that removes any leading and trailing space and tab characters.
- That will ensure that we do not add any items consisting of only spaces or tab characters.

Removing Items

- Similarly, to remove an item from the Grocery List, we will require the user to select the item from the list and then press the **Remove** button.
- Let us write an event handler for when the **Remove** button is pressed.
 - It will need to get the index of the selected item from the list and call the model's **remove()** method using this index. Then the view should be updated.
- To get the selected item from the list, we can look in the JAVA API and determine that the **ListView** method we need to call is **getSelectionModel().getSelectedIndex()**.
- This will return **-1** if nothing is selected, so we should handle that.
- So we should add this method to the controller (i.e., the main application):

```
// The Remove Button event handler  
private void handleRemoveButtonPress() {  
    int index = view.getList().getSelectionModel().getSelectedIndex();  
    if (index >= 0) {  
        model.remove(index);  
        view.update();  
    }  
}
```


Removing Items

```
// Add the following to controller's constructor  
view.getRemoveButton().setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent actionEvent) {  
        handleRemoveButtonPress();  
    }  
});
```

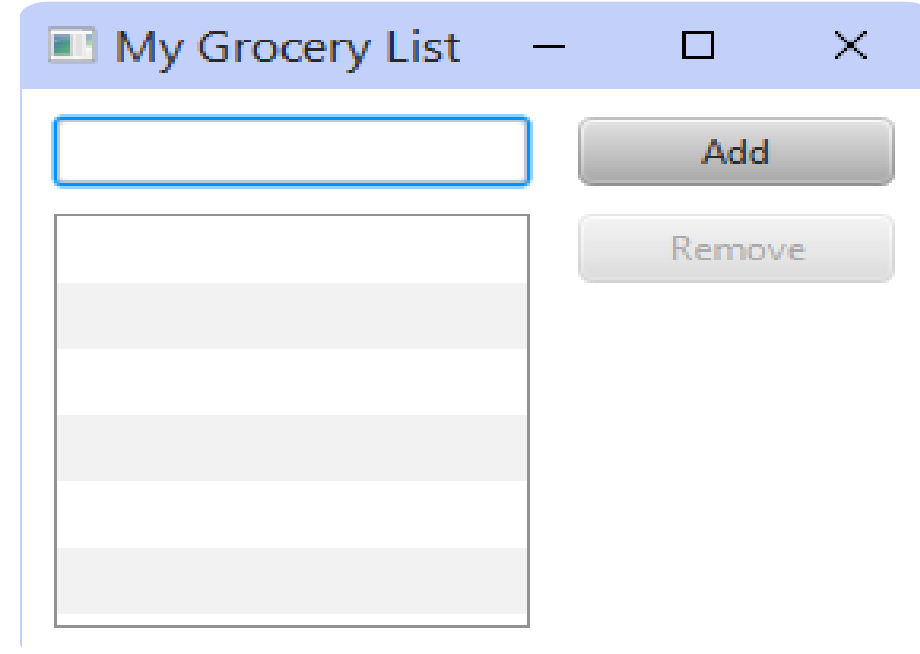
- This completes the basic functionalities for the code.

Extra's : Disabling the remove button

- It would be a good idea to disable the **Remove** button when nothing has been selected in the list.
- That way the user knows visually that the **Remove** operation is not valid until something is selected.
- It is a good form of feedback to the user and it makes the user interface more intuitive to use.
- Since this is simply a visual change, we could simply add a line to the **update()** method in the view class.

Extra's : Disabling the remove button

```
public void update() {  
    // Create and return a new array with the  
    // exact size of the number of items in it  
    String[] exactList = new String[model.getSize()];  
    for (int i=0; i<model.getSize(); i++)  
        exactList[i] = model.getItems()[i];  
  
    groceryList.setItems(FXCollections.observableArrayList(exactList));  
  
    // enable/disable the remove button accordingly  
    removeButton.setDisable(groceryList  
                            .getSelectionModel()  
                            .getSelectedIndex() < 0);  
}
```



Extra's : Disabling the remove button

- Logically, this code will disable the button if the selected index in the list is -1 (i.e., when nothing is selected).
- If we start up the application, the **Remove** button will be disabled properly since likely, nothing has been selected in the list.
- When should it be re-enabled ?
 - According to our **update()** method, whenever something is selected from the list it will be enabled.
- However, we need to ensure that the **update()** method is called when the user selects something from the list.
- Whenever the user clicks in the list, we can simply update the view to ensure that the **Remove** button is re-enabled.
- Here is the simple event handler to put into the controller class

```
// The ListView selection event handler
```

```
private void handleListSelection() { view.update(); }
```

Extra's : Disabling the remove button

- There are a few ways to cause this to occur. The simplest is to add a **mousePressed** event handler to the **ListView**. Again, we plug it in by adding this to the constructor:

```
view.getList().setOnMousePressed(new EventHandler<MouseEvent>() {  
    public void handle(MouseEvent mouseEvent) {  
        handleListSelection();  
    }  
});
```

- However, a minor issue arises in that when we update the list by repopulating its contents, the currently selected item becomes unselected.
- We can fix this by remembering what is selected before we update the list and then re-select that item again afterwards.
- In the view, we can alter the **update()** method to do this:

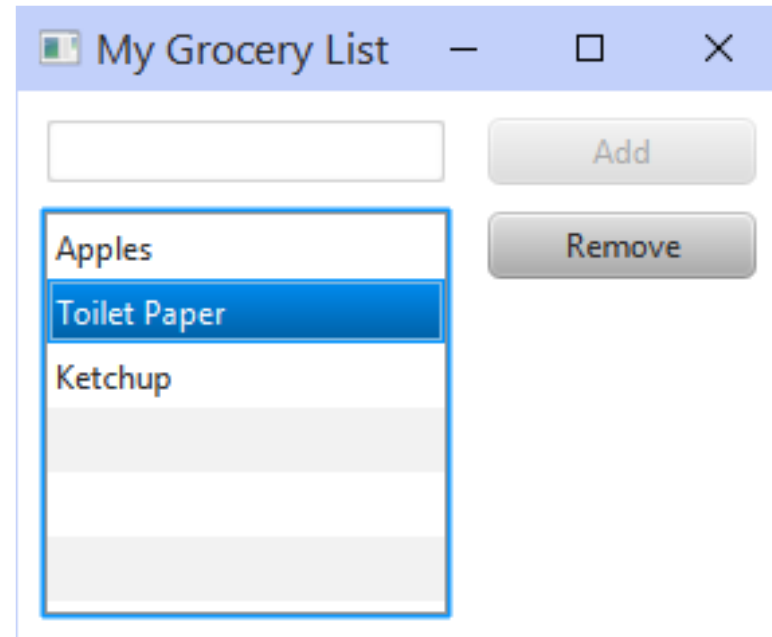
Extra's : Disabling the remove button

```
public void update() {  
    // Create and return a new array with the  
    // exact size of the number of items in it  
    String[] exactList = new String[model.getSize()];  
    for (int i=0; i<model.getSize(); i++)  
        exactList[i] = model.getItems()[i];  
    int selectedIndex = groceryList.getSelectionModel()  
        .getSelectedIndex();  
    groceryList.setItems(FXCollections.observableArrayList(exactList));  
    groceryList.getSelectionModel().select(selectedIndex); //  
    enable/disable the remove button accordingly  
    removeButton.setDisable(groceryList  
        .getSelectionModel()  
        .getSelectedIndex() < 0);  
}
```

Extra's : Disabling the Add button

- Finally, it would also be a good idea to disable the **Add** button when nothing is typed in the text field.
- To do this, we need to add a single line to the **update()** method in the view that is similar to the one we added to enable/disable the **Remove** button. However, this time we look at the text in the text field:

```
public void update () {  
    // ...  
    // enable/disable the Add button accordingly  
    addButton.setDisable(newItemField.getText().trim().length() <= 0);  
}
```



Extra's : Disabling the Add button

- Of course, once again we need to have the button re-enabled when the user starts typing into the text field.
- We need an event that occurs when the user types into a text field. Again the event handler is simple:

```
// The TextField typing event handler
```

```
private void handleTextEntry() { view.update(); }
```

- As with the **ListView** selection event handling, there are a few ways to plug in the event handler for the **TextField**.
- The simplest way is to handle a **KeyReleased** event

Extra's : Disabling the Add button

- So, whenever the user types a character into the text field (i.e., a character or delete, or ENTER, etc...), then the event will be called.
- However, take note that it will not be called if the user cuts or pastes something into the text field via a mouse operation.
- Also, a **KeyPressed** event will not work here because it will generate the event before the typed character is part of the text field's data, so we will always be missing one character when checking the length of the String.
- A **KeyReleased** event will be generated AFTER the character has been added to the text field, so this is fine.

```
view.getNewItemField().setOnKeyReleased(new EventHandler<KeyEvent>() {  
    public void handle(KeyEvent keyEvent) {  
        handleTextEntry();  
    }  
});
```

Extra's : Clearing the Text Field

- One final alteration to the program would be to clear the text field after an item has been added.
- Otherwise, after each item has been added, the user will have to delete the text before adding the next item. This can be tedious.
- To accomplish this, we simply add one more line to the **Add** button event handler to clear the text:

```
private void handleAddButtonPress() {  
    String text = view.getNewItemField().getText().trim();  
    if (text.length() > 0) {  
        view.getNewItemField().setText("");  
        model.add(text);  
        view.update();  
    }  
}
```