**CS 1302 – Lab 16**

To complete this tutorial, you must have the *e(fx)clipse* plugin to create JavaFX projects.

This is a tutorial on writing event handlers for *Graphical User Interfaces* (Gui). There are 7 stages to complete this lab:

|  |  |
| --- | --- |
| **Stage** | **Title** |
| 1 | Programming an Event Handler |
| 2 | Accessing a ComboBox in an Event Handler |
| 3 | Accessing a ListView in an Event Handler |
| 4 | Accessing a set of CheckBoxes in an Event Handler |
| 5 | Accessing a set of RadioButtons in an Event Handler |
| 6 | Maintaining State in a GUI Application |
| 7 | Example |

To make this document easier to read, it is recommended that you turn off spell checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”
3. **Programming an Event Handler**
4. **Read (no action required)** –
5. An *event handler* is code that is run when the user takes some action on a Gui, for example, when a button is pressed. The technique we use is to write *inner class event handlers.* Some other techniques, which are similar are: anonymous class, stand-alone class, and lambda expression event handlers.
6. Suppose we want to write a Gui that works as shown on the right. Assuming the Gui is written, there are two things to do:
7. Write a class that implements the *EventHandler* interface. The *EventHandler* interface requires a *handle* method:



1. *Register* the event handler with thebutton.



Then, when the button is pressed, the *handle* method is called.

1. Our event handler is an *inner class event handler.* An *inner class* is a class inside another class. An inner class can use all members of the enclosing class (even private). Note the figure on the right.
2. The *TextField* and *TextArea* must be declared as instance variables so that they can be accessed in the event handler. Note that the event handler does not use the *Label* nor the *Button*, so they could have been declared as local variables in the *buildGui* method. It might be useful to declare all controls as instance variables to avoid confusion.
3. Do the following:
4. Download and unzip the zip file associated with this lab.
5. Open Eclipse in a workspace.
6. Create a JavaFX project with the name, *lab16\_lastName*
7. Drag all files that were unzipped above into the *application* package in Eclipse. It will ask to overwrite *application.css*; answer, “yes”.
8. Open *Main1* and run. The GUI above should be displayed, but the event handler has not been coded yet.
9. Add the inner class event handler below to the *Main1* class. Remember that it goes inside the class, at the same level as a method or instance variable.

**private** **class** HelloWorldButtonEventHandler

 **implements** EventHandler<ActionEvent> {

 @Override

 **public** **void** handle(ActionEvent e) {

 String name = txfName.getText();

 txaMessage.setText(name + ", Hello World!");

 }

}

1. Register the event handler with the button by adding this line of code directly below the creation of the button in *buildGui*.

btnHelloWorld.setOnAction(**new** HelloWorldButtonEventHandler());

The result will look like this:

**private** Pane buildGui() {

 ...

 btnHelloWorld = **new** Button("Hello World");

 btnHelloWorld.setOnAction(**new** HelloWorldButtonEventHandler());

 grid.add(btnHelloWorld, 0, 1);

 ...

}

1. Run, and verify that the event handler is working as expected.
2. **Accessing a ComboBox in an Event Handler**
3.  **Read (no action required)** –
4. Consider the GUI on the right. The user selects an interest rate from the *ComboBox,* enters a balance, presses the button then a message is displayed showing the new balance after the interest is applied.
5. Thus, the button’s event handler needs to access the *selected value* from the *ComboBox* as well as the value in the *TextField*. Thus, these two must be declared as instance variables.
6. The selected value of the *ComboBox* is obtained through its *getValue* method. For example, if the *ComboBox’s* name is *cmbInterestRate*, then we can retrieve this value with this line of code in the event handler:

String strIntRate = cmbInterestRate.getValue();

Note, however, in the example above, that *strInterestRate=”3.0%”*. Thus, we need to strip off the “%”:

strIntRate = strIntRate.substring(0,strIntRate.length()-1);

And, we need to convert it to a double (and divide by 100 to make it a decimal) before doing computations with it.

**double** intRate = (Double.*parseDouble*(strIntRate))/100.0;

1. Open *ComboBoxExample* and run. The GUI above should be displayed, but the event handler has not been coded yet.
2. Add the inner class event handler below to the *ComboBoxExample* class. Study the code to understand what it is doing.

**private** **class** CalculateInterestEventHandler **implements** EventHandler<ActionEvent> {

 @Override

 **public** **void** handle(ActionEvent event) {

 String strIntRate = cmbInterestRate.getValue();

 // Remove "%" from the end

 strIntRate = strIntRate.substring(0,strIntRate.length()-1);

 **double** intRate = (Double.*parseDouble*(strIntRate))/100.0;

 **double** balance = Double.*parseDouble*(txfBalance.getText());

 **double** interest = balance\*intRate;

 **double** newBalance = balance + interest;

 String message = String.*format*("Old balance=$%,.2f\n" +

 "Interest=$%,.2f\nNewBalance=$%,.2f", balance, interest, newBalance);

 txaMessage.setText(message);

 }

}

1. Register the event handler with the button by adding this line of code directly below the creation of the button in *buildGui*.

btnCalcInterest.setOnAction(**new** CalculateInterestEventHandler());

The result will look like this:

**private** Pane buildGui() {

 ...

 Button btnCalcInterest = **new** Button("Calculate Interest");

 btnCalcInterest.setOnAction(**new** CalculateInterestEventHandler());

 grid.add(btnCalcInterest, 1, 3);

 ...

}

1. Run, and verify that the event handler is working as expected.
2. **Accessing a *ListView* in an Event Handler**
3.  **Read (no action required)** –
4. Consider the GUI on the right. The user selects any number of items from the *ListView,* presses the button then a message is displayed showing the items that were selected.
5. The selected items in the *ListView* are obtained with this line of code:

List<String> allItems = lvwInterests.getSelectionModel().getSelectedItems();

Then, we can loop over the list to access each item:

String interests = "";

**for**(String interest : allItems) {

 interests += interest + ", ";

}

1. Open *ListViewExample* and run. The GUI above should be displayed, but the event handler has not been coded yet.
2. Add the inner class event handler below to the *ListViewExample* class. Study the code to understand what it is doing.

**private** **class** ProcessEventHandler **implements** EventHandler<ActionEvent> {

 @Override

 **public** **void** handle(ActionEvent event) {

 String interests = "";

 List<String> allItems = lvwInterests.getSelectionModel().getSelectedItems();

 **for**(String interest : allItems) {

 interests += interest + ", ";

 }

 txaMessage.setText(interests);

 }

}

1. Register the event handler with the button by adding this line of code directly below the creation of the button in *buildGui*.

btnProcess.setOnAction(**new** ProcessEventHandler());

1. Run, and verify that the event handler is working as expected.
2. **Accessing a set of *CheckBoxes* in an Event Handler**
3.  **Read (no action required)** –
4. Consider the GUI on the right. The user selects any number of check boxes*,* presses the button then a message is displayed showing the items that were selected.
5. Note that we have individually named each *CheckBox* in a helper method to build the *HBox* that holds them.

**private** Pane buildFoodSelection() {

 ckbBurger = **new** CheckBox("Burger");

 ckbDrink = **new** CheckBox("Drink");

 ckbFries = **new** CheckBox("Fries");

 HBox hbxFood = **new** HBox();

 hbxFood.getStyleClass().add("h\_or\_v\_box");

 hbxFood.getChildren().addAll(ckbBurger,ckbDrink,ckbFries);

 **return** hbxFood;

}

1. In the *handle* event handler, we simply check the *isSelected* property of each *CheckBox*:

**if**(ckbBurger.isSelected()) {

 cost += 5.99;

 order += "Burger";

}

**if**(ckbDrink.isSelected()) {

 cost += 1.99;

 order += ", Drink";

}

**if**(ckbFries.isSelected()) {

 cost += 1.49;

 order += ", Fries";

}

1. Open *CheckBoxExample* and run. The GUI above should be displayed, but the event handler has not been coded yet.
2. Add the inner class event handler below to the *ListViewExample* class. Study the code to understand what it is doing.

**private** **class** ProcessEventHandler **implements** EventHandler<ActionEvent> {

 @Override

 **public** **void** handle(ActionEvent event) {

 **double** cost = 0.0;

 String order = "Order:\n";

 **if**(ckbBurger.isSelected()) {

 cost += 5.99;

 order += "Burger";

 }

 **if**(ckbDrink.isSelected()) {

 cost += 1.99;

 order += ", Drink";

 }

 **if**(ckbFries.isSelected()) {

 cost += 1.49;

 order += ", Fries";

 }

 String totCost = String.*format*("\nTotal: $%,.2f\n", cost);

 order += totCost;

 txaMessage.setText(order);

 }

}

1. Register the event handler with the button by adding this line of code directly below the creation of the button in *buildGui*.

btnProcess.setOnAction(**new** ProcessEventHandler());

1. Run, and verify that the event handler is working as expected.
2. **Accessing a set of *RadioButtons* in an Event Handler**
3.  **Read (no action required)** –
4. Consider the GUI on the right. The user selects a *RadioButton,* presses the button then a message is displayed showing the items that were selected.
5. Remember that we need to associate each *RadioButton* with a *ToggleGroup* to force them to work as a group (only one *RadioButton* can be selected). This is shown below in the helper method to build the *VBox* that holds them (and the *Button*). As we will see, the event handler only needs access to the *ToggleGroup* to determine which one is selected.

**private** Pane buildDiningChoice() {

 tGrpDiningChoice = **new** ToggleGroup();

 rbDineIn = **new** RadioButton("Dine In");

 rbDineIn.setSelected(**true**);

 rbDineIn.setToggleGroup(tGrpDiningChoice);

 rbTakeOut = **new** RadioButton("Take Out");

 rbTakeOut.setToggleGroup(tGrpDiningChoice);

 rbDelivery = **new** RadioButton("Delivery");

 rbDelivery.setToggleGroup(tGrpDiningChoice);

 Button btnProcess = **new** Button("Process");

 btnProcess.setOnAction(**new** ProcessEventHandler());

 VBox vbxDiningChoice = **new** VBox();

 vbxDiningChoice.getStyleClass().add("h\_or\_v\_box");

 vbxDiningChoice.getChildren().addAll(rbDineIn,rbTakeOut,rbDelivery,btnProcess);

 **return** vbxDiningChoice;

}

1. In the *handle* event handler, we use the *getSelectedToggle* methodof the *ToggleGroup* to return the *RadioButton* that was selected.

RadioButton rad = (RadioButton)tGrpDiningChoice.getSelectedToggle();

Next, we get the text of the radio button:

String choice = rad.getText();

Then, we use that text to determine which radio button was selected:

String message = "";

**switch**(choice) {

 **case** "Dine In" : message = "Glad you are dining in with us";

 **break**;

 **case** "Take Out" : message = "Meet you at the window";

 **break**;

 **case** "Delivery" : message = "We will have it there shortly";

}

txaMessage.setText(message);

1. Open *RadioButtonExample* and run. The GUI above should be displayed, but the event handler has not been coded yet.
2. Add the inner class event handler below to the *RadioButtonExample* class. Study the code to understand what it is doing.

**private** **class** ProcessEventHandler **implements** EventHandler<ActionEvent> {

 @Override

 **public** **void** handle(ActionEvent event) {

 RadioButton rad = (RadioButton)tGrpDiningChoice.getSelectedToggle();

 String choice = rad.getText();

 String message = "";

 **switch**(choice) {

 **case** "Dine In" : message = "Glad you are dining in with us";

 **break**;

 **case** "Take Out" : message = "Meet you at the window";

 **break**;

 **case** "Delivery" : message = "We will have it there shortly";

 }

 txaMessage.setText(message);

 }

}

1. Register the event handler with the button by adding this line of code directly below the creation of the button in *buildDiningChoice* helper method.

btnProcess.setOnAction(**new** ProcessEventHandler());

1. Run, and verify that the event handler is working as expected.
2. **Maintaining State in a GUI Application**
3. Open *ShapeGenerator* and run. The GUI will appear as on the right. Do the following:
4. There should be data in the text field (*e.g. 2.4 3.5 4.6*). Press “Create Shape” and a Triangle will be created.
5. Select the “Rectangle” radio button and type in the width and height of a rectangle, separated by a space (*e.g* 5.5 2.3) and the press “Create Shape”
6. Create a circle by supplying the radius.
7. Press the “Show All” button. Note that it says that it needs to be implemented.
8. **Read (no action required)** –
9. The code uses an abstract *GeometricObject* class with abstract methods, *getArea* and *getPerimeter*. It also uses three subclasses: *Triangle, Rectangle,* and *Circle*.
10. Display the *CreateShapeEventHandler* and quickly examine the code. We’ll describe it next.
11. The event handler uses the radio button to determine which shape to create using the parameters the user has typed in. To get the parameters, it is using the *split* method of the *String* class which we will study in a later chapter. However, I’ll quickly explain. This statement, of course, simply puts the input into a string:

String input = txfParameters.getText();

Then, the *split* breaks the string in “numbers” by removing any “,” or any spaces “\\s” and stores them in an array.

String[] parameters = input.split("[,\\s]+");

For example:

String input = “2.4 3.4 5.1”;

String[] parameters = input.split("[,\\s]+");

Would mean that: parameters[0]=”2.4”, parameters[1]=”3.4”, parameters[2]=”5.1”

1. Note that the *shape* variable is a local variable and so it dies when the event handler ends. What we would like to do is remember the shapes as they are added. We call this, *maintaining state.* This is simply done by:
2. Adding an instance variable to the class to hold *GeometricObjects*

**protected** ArrayList<GeometricObject> shapes = **new** ArrayList<>();

1. In the event handler, after the *case* statement, add this line to put the newly created shape in the list:

shapes.add(shape);

1. Once we have done this, then we can program the *ShowAllEventHandler* to loop over the list and display all the shapes.
2. Do the following:
3. Add this instance variable to the class:

**protected** ArrayList<GeometricObject> shapes = **new** ArrayList<>();

1. Add this line immediately after the *case* statement in the *CreateShapeEventHandler*:

// Add to collection

shapes.add(shape);

1. Replace the line of code in the *handle* method of the *ShowAllEventHandler* with:

String msg = "All Shapes:\n";

**int** i=1;

**for**(GeometricObject shape : shapes) {

 msg += String.*format*("%4d. %s\n", i++, shape);

}

txaMessage.setText(msg);

// Erase fields

txfParameters.setText(**null**);

1. Run, add some shapes, and then use the “Show All” button and verify that it is working.
2. **You will add a feature to the program by writing your own code. Do the following:**
3. Add a “Reset” button to the right of the “Show All” button. Steps:
4. Create an instance variable for the new button
5. Find the *buildButtonRow* method.
6. Create a new button and add it to the *hBox*.
7. Add an event handler that deletes all the shapes in the list and displays a message confirming this. Steps:
8. Copy the *ShowAllEventHandler* and paste.
9. Change the name of this new method to *ResetEventHandler*.
10. Delete everything in the *handle* method except the next-to-last line: *txaMessage.setText(msg);*
11. Write a message to display.
12. Above this, write code to clear all the shapes from the *shapes* list. You can do this in one line of code.
13. Register the event handler with the button. Steps:
14. Copy these two lines and paste. Then change to reflect the Reset button name and event handler

CreateShapeEventHandler btnEventHandler = new CreateShapeEventHandler();
btnCreateShape.setOnAction(btnEventHandler );

1. Run, add some shapes, show all, then reset, then show all to verify that it is working.

**Submission**

1. Do the following
2. Zip all the folders (packages) under the *src* folder into a zip file named: *lab16\_lastname.zip*
3. Upload your zip file to the *lab16* dropbox in Blazeview.

**You are done!**