ECE25100: OBJECT ORIENTED PROGRAMMING

Note 10 _ Event Handling

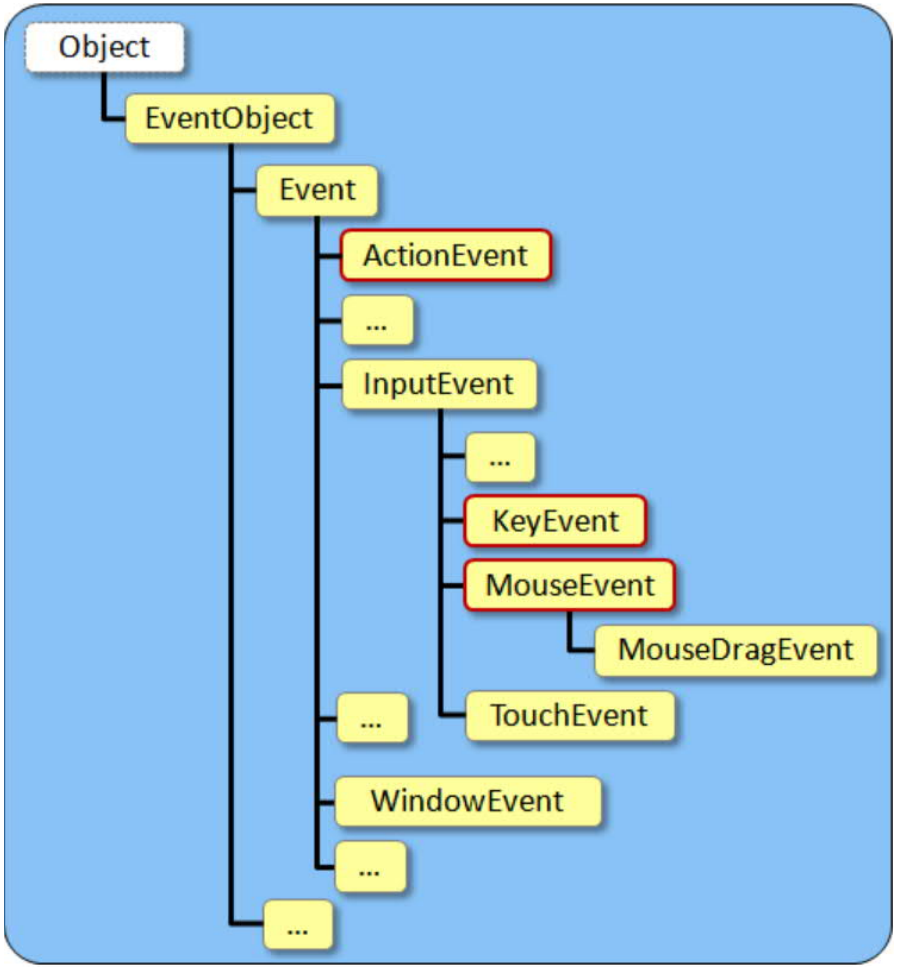
Instructor: Xiaoli Yang

Events and Event Handlers

- How to allow the user to interact with the window and make things happen?
  - An event is something that happens in the program based on some kind of triggering input which is typically caused (i.e., generated) by user interaction such as pressing a key on the keyboard, moving the mouse, or pressing a mouse button.
  - The source of an event is the component for which the event was generated (i.e., when handling button clicks, the Button is the source).
  - An event handler is a procedure that contains the code to be executed when a specific type of event occurs in the program.
- write specific event handlers in order for our application to respond to button clicks, allow selecting items from a list, allow typing into a text field

Events and Event Handlers

- In JAVA FX, all Events are represented by a distinct class. There are many kinds of events, each having its own unique class. Here is a partial hierarchy showing just a few of the events in the class hierarchy. We will look at the three highlighted in red.



Events and Event Handlers

These events are generated when the user interacts with the user interface as follows:

1. The user causes an event by clicking a button, pressing a key, selecting a list item etc..
2. Events are generated
3. The appropriate event handler is called.
4. The event handling code changes the model in some way.
5. The user interface is updated to reflect these changes in the model.

Events and Event Handlers

- In JAVA FX, you must first identify the types of events that you want to handle.Then you need to write the appropriate event handlers.
- For each event, there is a corresponding interface in JAVA with a list of methods that you can write in order to handle the appropriate event in a meaningful way.
- The table lists three of the commonly-used events along with the list of methods that you may implement to handle the kind of event that you are interested in.

Events and Event Handlers

Event	Method to Implement

ActionEvent - generated when button pressed, menu item selected, enter key pressed in a text field or from a timer event	setOnAction(new EventHandler() {
	  public void handle(ActionEvent actionEvent) {
	  // ...
	}});

KeyEvent - generated when pressing and/or releasing a key while within a component.	setOnKeyPressed(new EventHandler<KeyEvent>() {
	  public void handle(KeyEvent keyEvent) {
	  // ...
	}});
	setOnKeyTyped(new EventHandler<KeyEvent>() {
	  public void handle(KeyEvent keyEvent) {
	  // ...
	}});
	setOnKeyReleased(new EventHandler<KeyEvent>() {
	  public void handle(KeyEvent keyEvent) {
	  // ...
	}});

Events and Event Handlers

Event	Method to Implment

MouseEvent - generated when pressing/releasing/clicking a mouse button, moving a mouse onto or away from a component, moving the mouse over a component, and dragging something over the component.	setOnMousePressed(new EventHandler<MouseEvent>() {
	public void handle(MouseEvent mouseEvent) {
	// ...
	}});
	setOnMouseClicked(new EventHandler<MouseEvent>() {
	public void handle(MouseEvent mouseEvent) {
	// ...
	}});
	setOnMouseReleased(new EventHandler<MouseEvent>() {
	public void handle(MouseEvent mouseEvent) {
	// ...
	}});

Events and Event Handlers

Event	Method to Implement

MouseEvent - generated when pressing/releasing/clicking a mouse button, moving a mouse onto or away from a component, moving the mouse over a component, and dragging something over the component.	setOnMouseEntered(new EventHandler<MouseEvent>() {
	public void handle(MouseEvent mouseEvent) {
	// ...
	}});
	setOnMouseExited(new EventHandler<MouseEvent>() {
	public void handle(MouseEvent mouseEvent) {
	// ...
	}});
	setOnMouseDragged(new EventHandler<MouseEvent>() {
	public void handle(MouseEvent mouseEvent) {
	// ...
	}});
	setOnMouseMoved(new EventHandler<MouseEvent>() {
	public void handle(MouseEvent mouseEvent) {
	// ...
	}});

Events and Event Handlers

- For example, if you want to handle a button press in your program, you need to write a handle() method as follows:

```
aButton.setOnAction(new EventHandler() {
    public void handle(ActionEvent actionEvent) {
        // Write your code in here
    }
});
```

- For every event, we must not only write the event handler but also register that event handler too.

Events and Event Handlers

- There is an anonymous class (i.e., a class with no name) that simply has a handle() method within it which we must write.
- The handle() method takes an ActionEvent as a parameter. This will allow us to access information from the event such as the source of the event. In this case, it is the button object that generated the event.
- Similar methods are written for events generated from key presses, mouse presses, etc. The only different is that the parameter type changes from ActionEvent to KeyEvent or MouseEvent.

Examples

1. SimpleEventTest
2. TwoButtonApp: write separate event handlers for each button
3. MultipleButtonApp: more buttons share the same event handler.
4. ToggleButtonsApp: not robust as the code simply extracts the style string and looks at the character at position 14 in the string. What if we have more than one style set? What if we have different colors?
5. ToggleButtonApp2
6. CalculatorApp: How to use TextField and Button
7. CalculatorTwoApp: how to use ToggleGroup and RadioButtons.

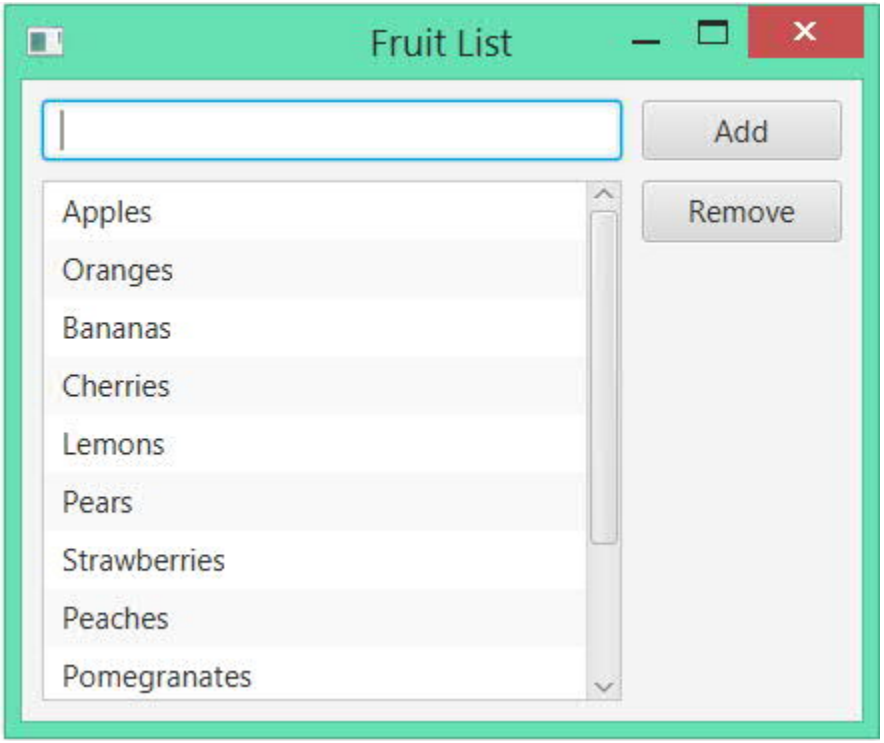
Model, View, and Controller

- The model classes deal with the business logic aspects of the application and the user interface is the "front end" which allows the user to interact with the model classes.
- We can further split the user interface classes into two portions called the view and the controller:
  - The view displays the necessary information from the model into a form suitable for interaction, typically a user interface element.
  - The controller accepts input from the user and modifies the model accordingly.

Model, View, and Controller

- This arrangement represents what is called the MVC software architecture and sometimes referred to as a software design pattern. There are 3 main advantages of using the MVC architecture:
  1. it decouples the models and views,
  2. it reduces the complexity of the overall architectural design
  3. it increases flexibility and maintainability of code. There are many ways to implement the MVC architecture in your programs.

Example


Fruit List
Add
Remove
Apples
Oranges
Bananas
Cherries
Lemons
Pears
Strawberries
Peaches
Pomegranates

Model

- Model include the classes that make up your application apart from the user interface components.
- It is a good idea to prepare your model so that you can interact with it in a simple and clean manner from your user interface.
- To finalize our model classes, we should decide what kinds of methods should be publically available so that the main application's user interface can access, modify and manipulate the model in meaningful ways.

Model

- What is the model in this application ? To figure this out, we just have to understand what "lies beneath" the user interface.
- What is it that we are displaying and changing ?
  - It is the list of items.
- Let us develop a proper model for this interface. We can call it ItemList and it can keep track of an array of Strings that represent the list.

Model

```
public class ItemList {
   public final int MAXIMUM_SIZE = 100;
   private String[] items;
   private int size;
   public ItemList() {
            items = new String[MAXIMUM_SIZE];
            size = 0;
   }
   public int getSize() { return size;}
   public String[] getItems() { return items;}
}
```

Model

- Change of the model by adding new item or remove selected item

```
public void add(String item) {
    if (size < MAXIMUM_SIZE)
            items[size++] = item;
}
```

```
public void remove(int index) {
  if ((index >= 0) && (index < size)) {
            for (int i=index; i<size-1; i++)
                        items[i] = items[i+1];
            size--;
  }
}
```

- ItemListTestProgram: without GUI.

View

```
import javafx.collections.FXCollections; import javafx.scene.control.*;
import javafx.scene.layout.Pane;
public class GroceryListView extends Pane {
   public GroceryListView() {
      TextField newItemField = new TextField();
      newItemField.relocate(10, 10);
      newItemField.setPrefSize(150, 25);
      Button addButton = new Button("Add");
      addButton.relocate(175, 10);
      addButton.setPrefSize(100, 25);
      Button removeButton = new Button("Remove");
      removeButton.relocate(175, 45);
      removeButton.setPrefSize(100, 25);
      ListView<String> groceryList = new ListView<String>();
      String[] groceries = {"Apples", "Oranges", "Bananas", "Toilet Paper", "Ketchup",
                            "Cereal", "Milk", "Cookies", "Cheese", "Frozen Pizza" };
      groceryList.setItems(FXCollections.observableArrayList(groceries));
      groceryList.relocate(10, 45);
      groceryList.setPrefSize(150, 150);
      getChildren().addAll(newItemField, addButton, removeButton, groceryList);}}
```

View

- Now, for the view to work properly it must refresh its look based on the most up-to-date information in the model.
- Therefore, we need a way of having the view update itself given a specific model.
- There are many ways to do this, but a simple way is to write a method called update() that will refresh the "look" of the view whenever it is called.
- To be able to update, the view must have access to the model. We can pass the model in as a parameter to the view constructor and store it as an attribute of the view:

View

```
public class GroceryListView extends Pane {
    private ItemList model;
    public GroceryListView(ItemList m) {
        model = m; // Store the model for access later
        ...
    }

 // This method is called whenever the model changes
    public void update() {
        // ... code for refreshing the view
    }
}
```

View

- Since the model attribute is simply a reference (i.e., a pointer in memory) to the ItemList, then any changes to the ItemList will also affect the model stored in this model instance variable.
- Our update() method may be as simple as replacing the entire ListView with the new items currently stored in the model.
- So, regardless of what changes take place in the ItemList model, we simply read the list of items and re-populate the ListView with the latest items.

View

```
public class GroceryListView extends Pane {
    private ItemList model;
    private ListView<String> groceryList;
    public GroceryListView(ItemList m) {
        model = m;
        ...
        groceryList = new ListView<String>();
        ...
    }

   // This method is called whenever the model changes
    public void update() {
        groceryList.setItems(FXCollections.observableArrayList(model.getItems()));
    }
}
```

View

- Notice how the ListView is now easily accessible in the update() method.
- One problem, however, is that our model's array is always of size 100 regardless of how many items have been placed in it.
- The setItems() method will end up making a list of 100 items in it ... leaving many blanks.
- We can add additional code here to make a new array (for display purposes) which has a length exactly equal to the size of the items array.

View

```
public void update() {
    String[] exactList = new String[model.getSize()];
    for (int i=0; i<model.getSize(); i++)
        exactList[i] = model.getItems()[i];
        groceryList.setItems(FXCollections.observableArrayList(exactList));
}
```

- Updated GroceryListView
- Test Model and View: GroceryListViewTestProgram2

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 develop our event handlers to handle the user interaction. It will handle all user input and then change the model accordingly ... updating the view afterwards.

Controller

- 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;
       private GroceryListView view;
       public void start(Stage primaryStage) {
              model = new ItemList();
              view = new GroceryListView(model);
              //…add event handler
              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); } }
```

Controller

- ALL of your event handlers should do:
  1. Change the Model
  2. Update the View.

```
view.getAddButton().setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent actionEvent) { handleAddButtonPress(); }
 });
```

```
private void handleAddButtonPress() {
        model.add(view.getNewItemField().getText());
        view.update();
}
```

Controller

- 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.

```
public void update() {
        addButton.setDisable(newItemField.getText().trim().length() <= 0);
}
```

Controller

- 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:

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

Controller

- 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. So, whenever the user types a character into the text field, then the event will be called.
- However, 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.

Controller

- A KeyReleased event will be generated AFTER the character has been added to the text field, so this is fine. Here is the code to plug in a KeyReleased event handler:

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

Controller

- The program also needs 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.

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

- The event handler for Remove button is similar to the one for Add button.
- Example: BroceryListApp

References

- Carleton University COMP 1005 and 1006:
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1405/Notes/
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/
- Armstrong Atlantic State University
  http://www.cs.armstrong.edu/liang/intro10e/
- Oracle Java tutorial:
  http://docs.oracle.com/javase/tutorial/
- MIT opencourseware
  https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/