**ISTE-121 Day05 – TextArea, Menus, Inner Classes**

**TextAreas**

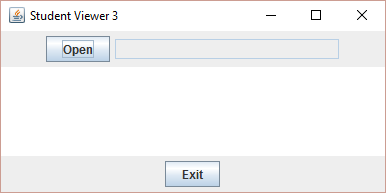
A companion to TextField is TextArea. The primary difference is that a TextField can hold only one line of text, while a TextArea can hold several lines.

* Make a copy of StudentViewer2, from last class, and name it StudentViewer3 (change the class name … then Save As).
* At the end of the doOpen method, call a new method, readData. This is a private method that accepts no parameters and returns nothing. **More on this later**.

Now, in place of the Labels and Textfields for the student’s name, ID, major, and GPA, declare a single JTextArea:

TextArea taData = new TextArea();

In the attributes, at the top of the class. Add this TextArea as the second child of the VBox. Run your program and you should see a window something like:



The big white area in the CENTER is the TextArea. When the file is Opened (in the end of the Open button … doOpen) you call readData. This method should read **all** of the data in from the file, a line at a time, in a while loop. Each time a line is read, add it to the JTextArea via:

jtaData.appendText(line + "\n");

The appendText method of TextArea sticks the string parameter on at the end of the text in the component. The newline is necessary, because nextLine (from Scanner) takes it off.

Run your program and open Students.txt from today’s download.

**Nicer Formatting**

Before we learn a little more about TextArea, let’s pretty up the output. To do this, we want to have the columns all line up nicely. This can be done with String.format. But the font of the TextArea is a proportional spaced font, so counting spaces in the output will not work for us (all characters do **not** use the same amount of space).

So, let’s change the font to a monospaced font (where all characters **do use** the same amount of space). To do this, we get the font for TextArea and create a new Font. A Font is a font name, a weight (bold, light, medium, etc.) and a size. We will use the normal weight and the same size as the TextArea uses by default, and just change the name to **"MONOSPACED"**.

Just before we add the JTextArea to the CENTER is where all this takes place:

Font currentFont = taData.getFont();  
 Font newFont =   
 Font.font("MONOSPACED", FontWeight.NORMAL,   
 currentFont.getSize());  
 taData.setFont(newFont);  
 root.getChildren().add(taData);

Then, before we add the data to the TextArea:

* Split the line into its fields, using String’s **split** function and a delimiter of :
* Use **String.format** to create a new string to display. I used 15 characters, left adjusted for the name, 5 spaces for the ID, 10 spaces, left adjusted for the major, and 4 spaces for the GPA.
* Display this new string, instead of the string read from the file, in the JTextArea … **don’t forget the newline**.

Get this to work before we move on …

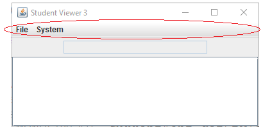
**Lots of Data**

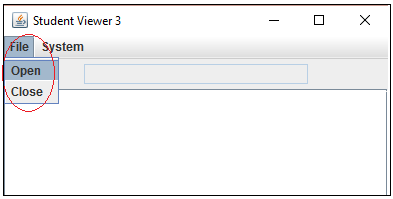
One last thing. What if the data is too long (or too wide) to fit in our TextArea. We would like scrollbars to handle this. TextArea automatically adds scrollbars, if the data so warrants.

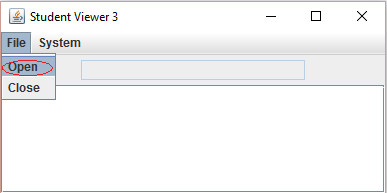
Now:

* Run your program and open Students.txt … the window should appear the same as before.
* Close Students.txt and open ManyStudents.txt. Now there are too many students to fit in the window, and a vertical scrollbar appears.
* Resize the window to make it taller, and the scrollbar will disappear.
* Resize it to make it narrower, and a horizontal scrollbar will appear.

**Menus**

There are three classes necessary to add menus to your program:  
 MenuBar

  
 Menu

  
 MenuItem

A program can have one menubar with several menus. Each menu can have several menuitems. As we will see, choosing a menuitem is just like clicking a button.

* Make a copy of StudentViewer3, named StudentViewer4.
* Remove the declaration of the Open/Close button and all references to it. Leave doOpen and doClose alone for now.
* Remove the Exit button and all references to it. In fact, remove the entire FlowPane at the bottom of the VBox.
* Now, declare a MenuBar as an attribute:  
   private MenuBar mbarBar = new MenuBar();
* Declare a Menu named "File" as an attribute  
   private Menu mnuFile = new Menu("File");  
  Similarly, declare a Menu named "System"
* Declare a MenuItem named "Open" as an attribute  
   private MenuItem miOpen = new MenuItem("Open");  
  Similarly, declare MenuItems named "Close", "About", and "Exit"
* Inside the constructor, after setting up the window, put the "Open" menu item in the "File" menu and call setOnAction for this menu item to our main class. Do the same with the "Close" menu item.  
  Also add the "About" and "Exit" menu items to the "System" menu and call setOnAction for them to our main class.  
   mnuFile.getItems().add(miOpen);  
   miOpen.setOnAction(this);
* Now, put the "File" menu in the menu bar. Also put the "System" menu in the menu bar.  
   mbarBar.getMenus().add(mnuFile);
* Finally, make the menu bar the menu bar for the window.  
   root = new VBox(mbarBar);  
  This means that the root pane for the window (the VBox in this case) must be created **after** the menubar is set up.

Temporarily comment out the body of handler and change it to:  
 MenuItem mi = (MenuItem)evt.getSource();  
 System.out.println(mi.getText());  
where evt is the ActionEvent parameter to handle. Run your program and select each menu item to see what name is passed into handle in each case.

Now, change handle back to its original version. Change the line  
 Button btn = (Button)evt.getSource();  
to  
 MenuItem mi = (MenuItem)evt.getSource();

Change the switch statement to:  
 switch(mi.getText()) {  
and add cases for "About" and "Exit". You should know what to do in the "Exit" case. In the "About" case use an Alert to display a message about the author. It should include the author’s name and email address. Feel free to use fictitious information if you like.

**A Couple of Improvements**

With our old version of StudentViewer2, you could not click Open twice without clicking Close in between, because of the way the button was renamed. However, with menus this is possible. Try it …

You get BOTH files opened and displayed in the text area. Change this so that if Open is clicked twice in a row (without a Close), the old information is deleted. You will want to call doClose from doOpen in this case, and in doClose you will want to use:

taData.setText("");

which sets the entire text area to the empty string. Declare the Scanner used to open and read the file as an attribute  
 Scanner scn = null;

When the file is opened, this variable is set to that file. You can check if a file is open by checking to see if

if(scn == null)

for this to work, be sure to set scn to null when you **close** the file. So, now, at the start of doOpen, if scn is **not** null, call doClose before proceeding.

Get this all to work before proceeding.  
  
**Event Handling Classes**

There are a number of ways that event handling classes (like ones that implement the EventHandler interface) can be implemented.

***Outer Classes***

// All of this in one file … MyApp.java   
// OR in two files … MyApp.java for the MyApp class and Listener.java   
// for the Listener class.   
//  
// If in separate files, the Listener class will be public.  
public class MyApp extends Application {  
 private Button btn1 = new Button("Button1");  
 . . .  
 private MenuItem mi1 = new MenuItem("MenuItem1");  
 . . .  
 public MyApp() {  
 . . .  
 Listener listen = new Listener(btn1, mi1, …);  
 btn1.setOnAction(listen);  
 mi1.setOnAction(listen);  
 . . .  
 } // end of constructor  
} // end of MyApp  
  
class Listener implements EventHandler<ActionEvent> {  
 private Button btn1;  
 private MenuItem mi1;  
  
 public Listener(Button \_btn1, MenuItem \_mi1, …) {  
 btn1 = \_btn1;  
 mi1 = \_mi1;  
 . . .  
 } // end of Constructor  
  
 public void handle(ActionEvent evt) {  
 String label;  
 Object src = evt.getSource();  
   
 if(src instanceOf Button)  
 label = ((Button)src).getText();  
 else if(src instanceOf MenuItem)  
 label = ((MenuItem)src).getText();  
   
 switch(label) {  
 case "Button1":  
 . . .  
 break;  
 case "MenuItem1":  
 . . .  
 break;  
 . . .  
 } // end of switch  
 } // end of actionPerformed  
} // end of Listener

* Can be in a separate file (**should** be public class, then)
* Can be in same file as Application class but not inside the { … } of the class. (should **not** be public then)
* Called ‘outer’ because they do **not** appear inside { … } of the Application class, or any other class, for that matter (only one public class per file, file must be same name as public class).
* Useful if:
  + Application class is getting too large for handle to be in it (becomes unreadable).
  + handle, itself, is becoming too large with too many sources of the event.
  + Need handlers for several kinds of event (ActionEvent, WindowEvent, …)
* The Listener class must implement each kind of event listener it will handle.
* The Listener class must have handler for each kind of event that can occur.
* Since this is **not** in the scope of the Application, or other classes, must have attributes of those classes (e.g., GUI components) passed into constructor so they may be manipulated by the handlers.

***Inner Classes (named)***

This is a class (like the above) but **is** declared **inside** the { … } of another class.

// All of this in one file … MyApp.java

public class MyApp extends Application {  
 private Button btn1 = new Button("Button1");  
 . . .  
 private MenuItem mi1 = new MenuItem("MenuItem1");  
 . . .  
 public MyApp() {  
 . . .  
 Listener listen = new Listener();  
 btn1.setOnAction(listen);  
 mi1.setOnAction(listen);  
 . . .  
 } // end of constructor

public class Listener implements EventHandler<ActionEvent> {  
 public void handle(ActionEvent evt) {  
 String label;  
 Object src = evt.getSource();  
   
 if(src instanceOf Button)  
 label = ((Button)src).getText();  
 else if(src instanceOf MenuItem)  
 label = ((MenuItem)src).getText();  
  
 switch(label) {  
 case "Button1":  
 . . .  
 break;  
 case "MenuItem1":  
 . . .  
 break;  
 . . .  
 } // end of switch  
 } // end of actionPerformed  
} // end of Listener

} // end of MyApp

* Now, the Listener class is inside the { … } of the MyApp class … it cannot be public.
* Because of scoping, the Listener class, and all of its methods, have access to all of the MyApp attributes, including GUI components.
* Otherwise, the same as an outer class

***Anonymous Inner Classes***

This is another type of inner class, but it has no name. Can only be used once because of this.

// All of this in one file … MyApp.java

public class MyApp extends Application {  
 private Button btn1 = new Button("Button1");  
 . . .  
 private MenuItem mi1 = new MenuItem("MenuItem1");  
 . . .  
 public MyApp() {  
 . . .  
 btn1.setOnAction(new EventHandler<ActionEvent>() {  
 public void handle(ActionEvent evt) {  
 // event handler for btn1 only  
 }  
 } // Matches '(' in new EventHandler<ActionEvent>() above  
 ); // end of setOnAction above  
  
 mi1.setOnAction(new EventHandler<ActionEvent>() {  
 public void handle(ActionEvent evt) {  
 // event handler for mi1 only  
 }  
 } ); // Typical way to end one of these  
 . . .  
 } // end of constructor  
} // end of MyApp

* Called ‘inner’ because it is entirely inside the ‘outer’ class (MyApp)
* NOTE: MyApp no longer implements EventHandler<ActionListener>
* Called anonymous because the class, and object, have no name
* Instantiates the EventHandler<ActionListener> interface directly
  + Overrides handle inside braces { … } following the new
* Since it is anonymous, can only be used for one component’s events.
* But, if the event handling is short, this is quick and easy.
* Look at StudentViewer5.java for an implementation that uses **all** anonymous inner classes.