**ISTE-121 Day04 – Event Driven Programming**

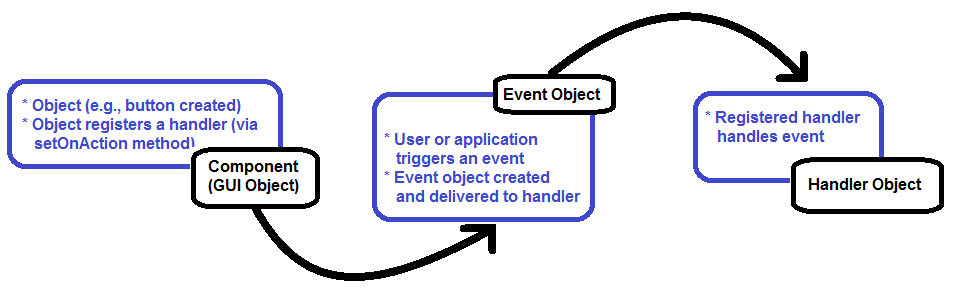
**Event Driven Programming**

When you write a GUI program, after the window appears, the program goes into a loop … not a loop you write… one that is built into the Java system … called an event loop. In this loop, it waits a short while, then checks for an ‘event’, handles it, and loops again.

An example of an event is a button click. If you click a button, and its setOnAction method has been called, then the event loop calls the **handle** method of the object passed to setOnAction, passing it a description of the event as an **ActionEvent** object.

There are many kinds of events that can occur to your program. Some caused by a human user (such as a button click), some caused by the OS (timers, windows changing state, etc), some caused by a program running on another computer (arrival of a message via the network). In the event loop, Java checks for all of these kinds of events and, if there is a **handler** registered for the event, calls the appropriate method.

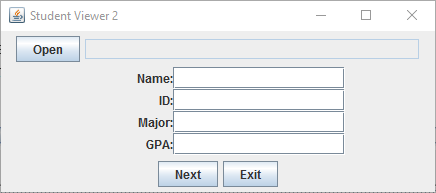
**Java Delegation Model for Event Handling**

****

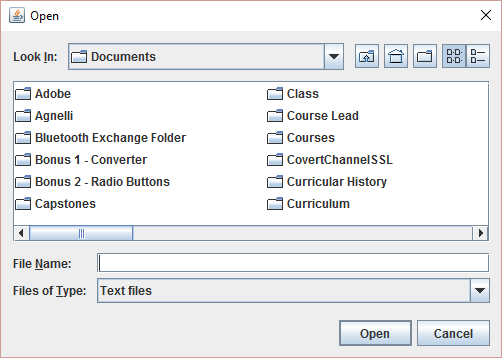
* **Handler object** may be of any class the implements the appropriate **EventHandler** interface
  + This event handler interface will be a generic interface dependent on a class that extends the **Event** class.
  + For example: EventHandler<ActionEvent>
  + There are others
    - EventHandler<InputEvent>
    - EventHandler<ScrollToEvent>
    - Etc.
* Handler object registers itself with the event loop (e.g., **setOnAction**)
* Often the class that implements the GUI is the handler for events
  + Allows the event handlers access to the private GUI components

**Example**

Write a class called StudentViewer2 (similar to StudentViewer from Day02). You might wish to start with your StudentViewer from earlier in the course. The GUI should look like:



When you click the Open button, a dialog like this should appear:



After the file is chosen, the Open button should become a Close button and the filename should appear in the text field to the right of the Close button. The rest of the program should work as in StudentViewer (the original version).

***How do we make this all work?***

For this project, I used a window of size 375x175.

The root layout was a VBox. Inside it were three panes:

* A FlowPane for the top (the Open button and the TextField to its right)
* A GridPane for the middle (the four labels with their TextFields)
* A FlowPane for the bottom (the Next and Exit buttons)

Add the main class (this) as the Handler for the buttons (including the new Open button).

Now … in actionPerformed, add two cases to the switch statement

case "Open":  
 doOpen();  
 break;  
 case "Close":  
 doClose();  
 break;  
The rest of the work involves writing doOpen and doClose.

In **doOpen**, we will instantiate a **FileChooser**, which paints the file selection dialog and manages it, and tells us what file is chosen. This is done in several steps, taken almost directly from the javadocs:

FileChooser fileChooser = new FileChooser();  
 fileChooser.setTitle("Open Student File");  
This instantiates a FileChooser object and sets the chooser’s title bar. We wish for this to show only text files. This is accomplished using a file extension filter:  
 fileChooser.getExtensionFilters().addAll(  
 new ExtensionFilter("Text Files", "\*.txt"),  
 new ExtensionFilter("All Files", "\*.\*"));  
  
Always include the ‘All Files’ option.

Next, we open the dialog, wait for the user to choose a file, and check the return value to see if the user clicked OK (as opposed to Cancel). In that event, we put the file name in the TextField for the filename (in the Top). **NOTE**: fileChooser.showOpenDialog() returns a File object … we will use this object to open the FileReader that is passed to the Scanner constructor.  
 File selectedFile = fileChooser.showOpenDialog(stage);  
 if (selectedFile == null) {  
 // Canceled  
 return;  
 }   
  
When this is done, we place the name of the chosen file in the tfFile TextField, open the file as a Scanner, and call doNext to read in the first line of the file.

// File selected. Display file name in text field  
 tfFile.setText(selectedFile.getAbsolutePath());  
   
 // Open the file  
 try {  
 scn = new Scanner(new FileInputStream(selectedFile));  
 doNext();  
 }  
 catch(Exception e) {  
 Alert alert = new Alert(AlertType.ERROR,   
 "Exception (" + e + ") opening file ... fatal.");  
 alert.showAndWait();  
 System.exit(1);  
 }  
The catch above catches any exceptions that might occur when the file is opened. It pops up a message in this event and terminates the program when the user acknowledges the message. This is done with the Alert mechanism covered in Day02.

Finally, change the name of the button to "Close".

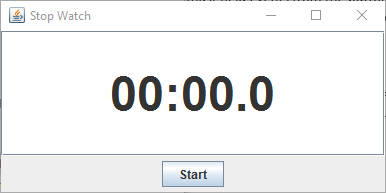
// Become the Close button  
 btnOpen.setText("Close");  
  
This trick causes the next click of the button to go to the "Close" case in **handle**.

***NOTES:***

* **handle** is a “dispatcher” … just calling the right method for the actual button clicked. The exception is the "Exit" case, where the code is so simple (1 line) we do it directly in **handle**.
* We introduced **FileChooser** to allow the user to browse the file system for the file s/he wishes and select it. For more details, see the javadocs.
* We introduced the idea of changing the name of a button so that a different method is called the next time it is clicked.

**In Class Exercise**

Write a new class called **StopWatch**. It should extend **Application** and implement **EventHandler<ActionEvent>**. The GUI should look like:



To accomplish this:

* Use VBox as the root layout.
* Add a TextField as the first child of the VBox and a FlowPane as the second child.
* To center the time, set the Alignment attribute of the TextField to Pos.CENTER. See the javadocs for details of how to do this.
* To make the time large, change its font. First:  
   Font currentFont = tfTime.getFont();  
   jtfTime.setFont(  
   Font.font(currentFont.getName(), FontWeight.BOLD, 48));
* This gets the font used by default for the TextField (tfTime.getFont()).
  + Then it creates a new font using the same font name, rendered in **bold** and in 48 point
  + Finally, it sets the font of the TextField to this new font

*For this to work: import javafx.scene.text.\*;*

The behavior of this program should be that, once started, it updates the time every 100 msec. To accomplish this, create and start a java.util.Timer object**. Use the full name java.util.Timer, as there are several Timer classes in the Java API**.

Create the timer and start it in the doStart method.

timer = new java.util.Timer();  
 timer.scheduleAtFixedRate(new MyTimerTask(), 0, 100);  
  
This will cause the **run** method of the class MyTimerTask to be called once every 100 msec to let you know that 100 msec has gone by.

For this to work, declare MyTimerTask as an **inner** class … a class declared **inside** your main class as in:

public class Stopwatch extends Application   
 implements EventHandler<ActionEvent> {  
 // Attributes are GUI components (buttons, text fields, etc.)  
 // are declared here.  
 private Stage stage; // The entire window, including title bar & borders  
 private Scene scene; // Interior of window

. . .

// NOTE: No **public** on inner classes  
 class MyTimerTask extends java.util.TimerTask {

public void run() {

}

}

}

Since this class (MyTimerTask) is inside the StopWatch class, it has access to all of StopWatch’s attributes. Declare a variable (long) called currentTime as an attribute of StopWatch and have MyTimerTask increment it by 100 each time it is called.

The run method above should also display the current time in the StopWatch’s TextField (tfTime) each time it is called.

Finally, the doStop method should call timer.cancel() where the timer is the one created inside of doStart … make this timer an attriubute of the StopWatch class and this should be easy.

***Notes:***

* Keep an attribute with the accumulated time for the timer (this should be of type **long**).
* Initialize accumulated time to 0L and the TextField to "00:00.0" (this means 0 minutes, 0 seconds, and 0 tenths of a second).
* On a Start click, clear the accumulated time (zero it out), and set the TextField to "00:00.0". Create the timer and start it. Change the name of the button to Stop.
* On each Timer event (in the run() method of MyTimerTask), increment the accumulated time by 100 (100 msec) and update the TextField. The TextField is formatted with:  
   String.format("%02d:%02d.%d",   
   minutes, seconds, tenthsOfSeconds)  
  it is up to you to figure out how to calculate these three values from the accumulated time.
* On a Stop click, cancel the timer. Change the name of the button back to Start.