# CS225L Lab 10: Event Handling

# Learning Outcomes

* ***Familiarization with browsing Java documentation and Googling***
* Use different types of event handlers to handle GUI events.
* Use three ways to write code that can handle events

# Pre-lab: Events

**What are events?**

Events are part of the model of event-driven programming. An event is basically some indication that something has happened in the program. When something happens, such as a mouse click or key press, the source object will fire that event. Other objects may listen for the firing of an event, and handle it by invoking some action when the event is heard.

**Creating event handlers**

Suppose we wanted to display some text to the console when a [Button](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Button.html) object is clicked. The method that we would use is [setOnMouseClicked](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html#setOnMouseClicked-javafx.event.EventHandler-) (EventHandler<? super MouseEvent>), inherited from the [Node](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html) class. The EventHandler<? super MouseEvent> part of the method signature means that we must give it an object that implements the EventHandler interface parameterized with MouseEvent or a superclass thereof. If you'd like to learn more about exactly what all of that means, look up [Generics](https://docs.oracle.com/javase/tutorial/java/generics/) (another aspect of Java that makes it particularly powerful, which you'll learn more about in SE320 if you take it); for this course, though, look at the examples below for the correct implementation:

**Explicit Classes**

To declare a class that matches the EventHandler<? super MouseEvent> parameter, our class declaration would look like this:

public class MyMouseClickHandler implements EventHandler < MouseEvent > {

@Override

public void handle ( MouseEvent e) {

// Do something when a MouseEvent happens

}

}

This can be declared in its own file, called MyMouseClickHandler.java, or as an inner class of our GUI class. Inner classes help consolidate your code, but they may only be used in your outer class. The declaration of an inner class may look like this:

public class MyGUIClass extends Application {

// outer class body here

class MyMouseClickHandler implements EventHandler < MouseEvent > {

@Override

public void handle ( MouseEvent e) {

// Do something when a MouseEvent happens

}

}

}

To add this handler to a button, the code would be as follows:

myButton.setOnMouseClicked(new MyMouseClickHandler());

When we implement the EventHandler interface, notice that we must overwrite the handle method. The contents of the handle method define what you want the program to do when the event occurs. For example, to print some text, our method would look like this:

public void handle ( MouseEvent e) {

System.out.println (" Button clicked !");

}

**Anonymous Inner Classes**

An anonymous inner class, as the name suggests, is an inner class that is not explicitly named. Recall the syntax for implementing a handler class. The code example below will show how to translate that to an anonymous class:

myButton.setOnMouseClicked (new ~~MyMouseClickHandler()~~);

~~public class MyMouseClickHandler implements~~ EventHandler < MouseEvent > () {

@Override

public void handle ( MouseEvent e) {

System.out.println (" Button clicked !");

}

} );

With all of the changes applied, it looks like this:

myButton.setOnMouseClicked (new EventHandler < MouseEvent >(){

@Override

public void handle ( MouseEvent e) {

System.out.println (" Button clicked !");

}

});

**Lambda Handler**

[Lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) are another new feature in Java 8. One of their uses is as [event handlers](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#lambda-expressions-in-gui-applications). Consider the above example. The handle method is given a MouseEvent object, which we've called e, that contains information about the event. This is passed to the method by the firing object. Using a lambda expression, we can reduce our code significantly while still maintaining a reference to the event object. The syntax of a lambda expression looks like this:

(arg0 , arg1 , ...) -> { statements }

You do not have to specify the variable types since the Java compiler can figure out what they are from the context. For example, a lambda in a setOnMouseClicked has to become an EventHandler<MouseEvent>, whose only method takes exactly one MouseEvent object. Additionally, if you only have one line of code in your statements, you may omit the braces and the semicolon ending the line. In our example GUI

application, we can reduce our anonymous inner class to a expression in the following way:

myButton.setOnMouseClicked (~~new EventHandler<MouseEvent>() {~~

~~@Override~~

~~public void handle(MouseEvent~~ e~~) {~~ ->

System.out.println (" Button clicked !");

}

});

Notice that the variable name e is kept, but all of the preamble is gone. With these changes made, the code reduces to one line:

myButton.setOnMouseClicked (e -> System.out.println (" Button clicked !" ));

**Lab Activities**

The EventsDemo.java file that you'll download has menu items and methods, but they aren't connected in any way. To clarify, when an action event is triggered by clicking a menu item, we currently don't have any code to handle it. As discussed in the pre-lab, there are several ways to handle events: explicit classes, anonymous inner classes, and lambda handlers. We will use each of those here to add functionality to the program.

**Instructions**

1. Import the lab files into Eclipse as an existing project.
2. Using the provided CircleHandler inner class, make the “Circle” menu item toggle the display of the circle based on whether it is selected or not. (Hint: this will use the setOnAction(...) method.) **(5 Points)**
3. Write a new anonymous inner class to have the “Line” menu item toggle the display of the line shape. **(5 Points)**
4. For the “Triangle” menu item, write a new [Lambda expression](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) that toggles the display of all 3 lines that make up the triangle shape. **(5 Points)**

*Note: If Eclipse fails to recognize lambda expressions, make sure that your JRE Compliance is set to at least 1.8 (Window -> Preferences -> Java -> Compiler). 1.8 compliance is only available in Eclipse Luna or higher.*

1. Using the provided ShowAllHandler inner class, make the “Show All” menu item show all shapes when it is pressed. **(5 Points)**
2. Convert the ShowAllHandler class to be able to handle either showing all items or clearing all items: **(5 Points)**
   1. Notice how ShowAllHandler uses a hard-coded value of true for all of the calls to setVisible? Instead, create a boolean field for ShowAllHandler (outside of the handle method), and replace all of the instances of true with that new variable.
   2. You should set your variable one time, when you create a new ShowAllHandler. Add a constructor to ShowAllHandler that takes one boolean argument. In that constructor, copy the argument to the variable you made in step 6a. (Remember how you did this in Lab 8?)
   3. This will break what you did in step 5, because you now have to provide an argument when you call new ShowAllHandler(). Replace that with new ShowAllHandler(true) for the “Show All” button.
   4. Reuse your modified ShowAllHandler class to make the “Clear all” menu item clear all of the shapes.
3. Using a lambda expression, make the “About. . . ” menu item under “Help” run the showAbout() method when clicked. **(5 Points)**
4. Using a lambda expression, make the “Close” menu item under “File” close the program. (Close the program by calling [Platform](https://docs.oracle.com/javase/8/javafx/api/javafx/application/Platform.html).exit() ). **(5 Points)**
5. Show your modified program to a lab instructor. If your program looks OK, go ahead and do the post-lab on the next page. (See example outputs in demo.mov)

**Lab Activities**

Include your edited EventsDemo.java file with your submission. Then, answer these questions in a separate document to include in the same submission. **(5 Points)**

1. What are three ways to write code that can handle events?
2. What are the differences between lambda expressions and classes? Why would you use a class instead of a lambda expression for an event handler?
3. Provide feedback:

(a) What is your lab section? (Section # or day of week)

(b) What did you like about the lab? What did you dislike?

(c) What would you change about this lab to improve it?

Total Points from Lab 10: **40 Points**