JavaFX events

Tamás Ambrus, István Gansperger

Eötvös Loránd University
ambrus.thomas@gmail.com

What events are

Events are notifications. They indicate that something has happened. This way:

- **view** can notify the **controller** of user actions,
- controller can invoke the **model** to execute some logic,
- controller changes the view based on the result provided by the model.

Kinds of events

Basically, everything a user can do on the UI has a corresponding event. Some notable examples:

- *KeyEvent*: key on the keyboard is pressed
- *MouseEvent*: mouse is moved or a button on the mouse is pressed
- *MouseDragEvent*: full mouse press-drag-release action is performed
- *ActionEvent*: GUI button is pressed, combo box is shown or hidden, or a menu item is selected

Kinds of events

Basically, everything a user can do on the UI has a corresponding event. Some notable examples:

- *KeyEvent*: key on the keyboard is pressed
- *MouseEvent*: mouse is moved or a button on the mouse is pressed
- *MouseDragEvent*: full mouse press-drag-release action is performed
- *ActionEvent*: GUI button is pressed, combo box is shown or hidden, or a menu item is selected

We can bind event handlers to GUI components in the controller and in the view.

Binding in the controller

```
Button btn = ...;
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("x");
    }
});
```

- create the GUI component
- call **setOnAction**
- define an *EventHandler* through the *handle* method

Anonymous classes

In Java, abstract classes and interfaces can be "instantiated" too, in a different way:

- you have to provide an implementation for **all** abstract methods
- you don't give a name to a class instance like this, hence it's anonymous
- this is a one-shot phenomenon

Anonymous classes are used often by programmers when they need a short, easy implementation of something abstract.

Lambda expressions

From Java 8 onwards, lambda expressions can be used too, on **Functional interfaces** (SAM).

Functional interface: has exactly one abstract method.

Our above code could become:

```
Button btn = ...;
btn.setOnAction(ev -> System.out.println("x"));
```

Lambda expressions

From Java 8 onwards, lambda expressions can be used too, on **Functional interfaces** (SAM).
Functional interface: has exactly one abstract method.

Our above code could become:

```
Button btn = ...;
btn.setOnAction(ev -> System.out.println("x"));
```

"The setOnAction method can only take an EventHandler. EventHandler is an interface whose only abstract method is handle, that takes an event. I'm gonna define this 1 parameter taking abstract method like this. This syntax cannot be misunderstood, I couldn't refer to another method with this definition."

Binding in the view

FXML:

```
<Button text="Don't click on me" onAction="#punishThisGuy" />
```

Controller:

```
@FXML
public void punishThisGuy() {
    System.out.println("You lost, get rekt");
}
```

- it can have 0 or 1 parameters (ActionEvent)
- any visibility is allowed

Event context

What is ActionEvent, KeyEvent, ... good for? These are context parameters.

They inform us about what element the event was fired on, or:

- by a KeyEvent which key was pressed
- by a MouseEvent how many mouse clicks were associated to which mouse button
- etc.

It's recommended to have a look at these classes as you may use often the information provided by them.

Closures

The body of a lambda expression or an anonymous class is called a closure. Inside the closure we can only access final parameters of the parent class or methods. Consider the following:

```
int i = 3;
loginButton.setOnAction(x -> System.out.println(i));
i = 4;
```

We get an error: *Local variable i defined in an enclosing scope must be final or effectively final.*

Closures

If we mark i as final then we get another error because we try to set it with $i = 4$ but if we remove that it works.

```
final int i = 3;
loginButton.setOnAction(x -> System.out.println(i));
```

In Java 8 we don't even have to write final because the compiler can figure out that the variable is effectively final.

How could we set i from within the lambda given these conditions?

E.g. Boxed

```
public class Boxed<T> {
    private T value;

    public Boxed(T value) { this.value = value; }
    public void set(T value) { this.value = value; }
    public T get() { return value; }
}
```

Now we can write:

```
Boxed<Integer> i = new Boxed<>(3);
loginButton.setOnAction(x -> System.out.println(i.get()));
i.set(4);
```

because *i* itself is *effectively* final.

References

- https://docs.oracle.com/javafx/2/events/jfxpub-events.htm
- https://docs.oracle.com/javase/9/docs/api/java/lang/FunctionalInterface.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html