Introduction to Object-Oriented Programming

JavaFX Events

Christopher Simpkins
chris.simpkins@gatech.edu

Outline

- Hello, JavaFX!
- Event-driven programming
- Hello, buttons!
- The observer pattern

Hello, JavaFX!

Here's a minimal JavaFX program:

```
import javafx.application.Application;
import javafx.scene.control.Label;
import javafx.scene.text.Font;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class HelloJfx extends Application {
    public void start(Stage stage) {
        Label message = new Label("Hello, JavaFX!");
        message.setFont(new Font(100));
        stage.setScene(new Scene(message));
        stage.setTitle("Hello");
        stage.show();
    }
}
```

See HelloJfx.java and the API documentation for JavaFX

javafx.application.Application

JavaFX programs include one class that extends `Application`, analogous to having a single class with a `main` method for console programs. When running an `Application` the JavaFX run-time:

1. Constructs an instance of the specified Application class
2. Calls the `init()` method for application initialization - **don't construct a `Stage` or `Scene` in `init()`**
3. Calls the `start(javafx.stage.Stage)` method
4. Waits for the application to finish, which happens when either of the following occur:
   - the application calls `Platform.exit()`
   - the last window has been closed and the `implicitExit` attribute on `Platform` is `true`, which is the default
5. Calls the `stop()` method - release resources obtained in `init()`

`init()` and `stop()` have default do-nothing implementaitons. To write a simple JavaFX program you create a subclass of `Application` and put GUI start-up code in the `start(javafx.stage.Stage)` method.

Setting the Stage

The JavaFX `Stage` class is the top level JavaFX container. A primary stage for your application is constructed by the JavaFX run-time and passed to your application in the `start(javafx.stage.Stage)` method:

```
@Override public void start(Stage stage) {
    Scene root = ...
    stage.setScene(new Scene(root));
    stage.setTitle("Hello");
    stage.show();
}
```

You construct a `Stage` for each window in your application, e.g., for dialogs and pop-ups, and add visual components to the stage using a scene graph.

Setting the Scene

The JavaFX `Scene` class is the container for all content in a scene graph. Every `Scene` has a root node, which may have children (which is why it's called a scene *graph*). In HelloJfx.java we simply used a `Label` control as the root node:

```
    @Override public void start(Stage stage) {
        Label message = new Label("Hello, JavaFX!");
        stage.setScene(new Scene(message));
        stage.setTitle("Hello");
        stage.show();
    }
```

A typical application will use as its root node a

- Group, typically used for graphics and animation components,
- a Region class for nodes that can be resized and styled with CSS (like UI controls or layout panes), or
- a Pane sublclass for laying out children according to some layout policy.

Event-Driven Programming

So far we've done structured sequential programming where the order of execution is controlled by the programmer. GUIs use event-driven programming:

- User is presented with options.
- User actions (and other actions) fire events.
- Event handlers execute in response to events.
- Order of execution is controlled by the order of events, which the programmer does not know in advance.

Hello, buttons!

```
public class HelloJfxButtons extends Application {

    int count = 0;

    @Override public void start(Stage stage) {
        Label counterLabel = new Label("Count: 0");
        Button incButton = new Button("Increment Count");
        incButton.setOnAction(event ->
            { counterLabel.setText("Count: " + (++count)); });

        VBox root = new VBox();
        root.getChildren().addAll(counterLabel, incButton);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Hello");
        stage.show();
    }
}
```

Note that HelloJfxButtons.java uses a VBox as its layout manager. We'll have more to say about layout in a future lecture.

Top-Level GUI Program Recipe

The HelloJfxButtons.java example demonstrates a simple recipe for JavaFX GUI programs:

1. Create UI controls

```
Label counterLabel = new Label("Count: 0");
Button incButton = new Button("Increment Count");
incButton.setOnAction(event ->
    { counterLabel.setText("Count: " + (++count)); });
```

2. Add UI controls to a parent node (Group, Region, or Pane)

```
VBox root = new VBox();
root.getChildren().addAll(counterLabel, incButton);
Scene scene = new Scene(root);
```

3. Set the stage with the scene graph and show it

```
stage.setScene(scene);
stage.setTitle("Hello");
stage.show();
```

The Observer Pattern in JavaFX

Three particpants in the observer pattern:

- An event publisher that fires events
- An event object that represent the event
- Event handlers that subscribe to event publishers and receive event objects

Practically speaking, firing an event means calling a method on event listeners. Let's look at a concrete example.

An Event Publisher: `javafx.scene.control.Button`

In `HelloJfxButtons.java` we set up an increment button like this:

```
Button incButton = new Button("Increment Count");
incButton.setOnAction(event ->
   { counterLabel.setText("Count: " + (++count)); });
```

- `Button`'s `setOnAction` method takes an object that implements the `javafx.event.EventHandler<ActionEvent>` interface.
  - `javafx.event.EventHandler<ActionEvent>` has one abstract method: `void handle(ActionEvent event)`
- When the button is pressed, the `void handle(ActionEvent event)` is invoked on the object passed to `setOnAction`, in this case an anonymous inner class that implements the `EventHandler` interface, which was instantiated by a lambda expression.

javafx.event.EventHandler<T>

```
@FunctionalInterface
public interface EventHandler<T extends Event>
        extends EventListener {

    /**
     * Invoked when a specific event of the type for which this
     * handler is registered happens.
     *
     * @param event the event which occurred
     */
    void handle(T event);

}
```

- `java.util.EventListener` is a tagging interface that all event listener interfaces must extend (an implementation detail you don't need to worry about).

Our Button Event Handler

Consider this alternate syntax for setting an event handler for our button (just to reinforce that a lambda expression is just syntax sugar for creating anonymous inner class instances of functional interfaces):

```
incButton.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
        counterLabel.setText("Count: " + (++count));
    }
});
```

- Our `EventHandler` captures the `counterLabel` local variable (which is effectively final) and the `count` instance variable
- When its `handle` method is called, it updates the count and (re-) sets the text on `counterLabel` with the new count

Three objects cooperating: a `Button`, a `Label`, and an `EventHandler` to tie them together.

Closing Thoughts

- Event-driven GUI programming requires a shift in thinking. Putting the user in control means you have to work harder to
  - handle order dependencies, e.g., by disabling buttons until certain actions are taken, and
  - guide the user, e.g., by following UI guidelines to maximize familiarity.
- Notice how the JavaFX framework contains classes and interfaces that you extend and implement - OOP in action.