Guide to JavaFX

Spring 2019

Contents

1 Introduction

JavaFX is an API (Application Programming Interface) that provides a set of graphics and media packages that allow us to design and customize client-facing applications. We will be using JavaFX to create a graphical user interface (GUI). A GUI is a specific type of user interface that allows people to interact with a program through icons and other visual elements. Another type of user interface that you may be familiar with, for instance, is the command-line interface.

2 How to Begin

Every basic JavaFX interface must consist of three main files: an FXML file, a `Controller` class (a scala file, in our case), and a `App` class (also a scala file). An FXML file defines the layout and visuals

of your user interface, while a `Controller` defines the functionality of each element in your FXML interface. The `App` class will allow you to bind the two together and run your final application. There are a couple of key JavaFX constructs that will be useful in setting up the framework for a user interface. These should be defined in your FXML file:

2.1 GridPane

The GridPane specifies the basic layout of an interface. The GridPane provides a common environment for each child element to exist within. GridPane organizes its `Nodes` (components) onto a grid of rows and columns. The child elements of a GridPane can be placed anywhere on the window and can span any number of rows and columns.

Note: JavaFX (and many other interfacing tools) measures screen coordinates using pixels with the top-left corner of the window as the origin and incrementing downwards and to the right.

2.2 ScrollPane

A ScrollPane allows you to display sections of content that the user can scroll through.

2.3 VBox

The VBox class creates a layout that orders its children in a single vertical column within the box. The VBox class inherits the `getChildren` method from the `Pane` class. The `getChildren` method returns an `ObservabileList` of `Nodes` (`ObservableList<Node>`). You can add JavaFX elements to the VBox using the `add` method and you can empty the VBox by using the `clear` method both inherited from the `List` interface.

2.4 HBox

The HBox operates in a similar way, with the distinction of arranging its children horizontally.

3 Rendering JavaFX Elements

Now that you have created a basic framework for your user interface, you want to be able to display a variety of different pages with different content requirements. Each of these HTML elements can be rendered into JavaFX objects and added as children to a VBox object. Here are a couple of basic classes that you can use when rendering a page:

3.1 Text

The Text class constructs an object that displays text to the user. You can use the `setText("text")` method to add your text and use the `setWrappingWidth(width)` method format your text to the dimensions of the interface.

3.2 Label

Labels are another option when rendering text in JavaFX. For basic implementations, the Label and Text classes are interchangable. In general, Text can be used to render text not associated with user input while Labels can be used to describe other input control elements.

3.3 Hyperlink

A hyperlink in JavaFX will render a link. You can use `setText` to set the text of your link. You will also need to set up an event handler for when the link is clicked. You can use the `setOnAction` to do this.

3.4 Button

Adding a Button element to your GUI will render a generic button. You can refer to the example given in the Section 5.1.

3.5 TextField

The TextField component allows users to enter a single line of plain text. You can set the prompt text using the `setPromptText` method and retrieve the text using the `getText` method. We will go into more details on connecting to a TextField object in Section 5.2.

All of the above elements can be added as children to a VBox. The VBox will then display its children when the GUI is run.

4 Application

Our main JavaFX application extends the imported abstract class `javafx.application.Application`.

Application contains an unimplemented method "start" that has a `Stage` object as a parameter.

```
class RunApplication extends Application {
  override def start(stage: Stage) {
  }
}
```

The stage is the actual window that will pop up on your screen when you open the browser. The stage contains the built-in minimize, full-screen, and close icons, as well as allows users to move the window around on their screen. Since you probably do not want the window opening and closing in the middle of your application, you will most likely only ever use one stage.

For the application to show anything, you should also set a `Scene`, which is the main container for all the elements that you have defined for a page. You will need three objects in order to do this:

- `Parent`: A form of Branch node and the base class for all nodes that have children in the scene graph.

- `FXMLLoader`: Loads an object hierarchy from an XML document.
- `Controller`: A class where the functionality of each page element is defined

The first step in setting the scene will involve loading your FXML file. You will need to create a `FXMLLoader` object and then call its load method to obtain the root (parent) node. The root node will be the outermost wrapper in your FXML. Using the loader, you can also call the `getController()` method to set up the controller.

```
class RunApplication extends Application {
  override def start(stage: Stage) {
    val loader = new FXMLLoader(getClass().getResource("page.fxml"))
    val par: Parent = loader.load().asInstanceOf[GridPane]
    val control : GUIController = loader.getController()
  }
}
```

Now that we have set up all the necessary structures, the last step is to actually display the page. The following methods may be helpful in displaying your scene:

- `setTitle(title : String)`: Allows you to set the title of the window
- `setScene(newScene : Scene)`: Allows you to set the scene
- `show`: Attempts to show the window by setting visibility to true

```
class RunApplication extends Application {
  override def start(stage: Stage) {
    val loader = new FXMLLoader(getClass().getResource("page.fxml"))
    val par: Parent = loader.load().asInstanceOf[GridPane]
    val control : GUIController = loader.getController()

    stage.setTitle("Guizilla")
    stage.setScene(new Scene(par,100,100))
    stage.show()
  }
}
```

At this point, you are ready to run your `Application`. Create an object that extends `App` and launch your application!

```
object RunApplication extends App {
  Application.launch(classof[RunApplication], args:_*)
}
```

5 Controller

In the previous section, we mentioned that we were going to create a controller object to define functionality for the scene. We want the controller to control several aspects included in the actual GUI and FXML. We have broken them down into different elements.

For example, say this is our FXML page:

```
<GridPane fx:controller="MainControl">
  <Button fx:id="myButton" onAction="#buttonAction"/>
  <Button fx:id="printButton" onAction="#printAction"/>
  <TextField fx:id="myTextfield"/>
</GridPane>
```

We make a class to link to the `fx:controller="MainControl"` tag in the FXML. It is empty for now, but below we will give a few examples of what to populate this class with.

```
class MainControl {

}
```

5.1 Button

The button control can contain text and/or a graphic. Whenever a button is pressed, the `onAction` method is invoked.

In our example, the button we have has an id of *myButton*. Our button, *myButton* also has an action, which can be referenced with *buttonAction*. First, to access the button, we have to make sure to create the variable corresponding to it in the FXML page.

```
class MainControl {
  @FXML private var myButton: Button = null
}
```

In the fxml file, an *onAction* is included. This means whenever this button is clicked (or if you changed the *onAction*, whatever you specified the *onAction* to be), you can tell your controller what its supposed to do. In the next example, the button exits the program.

```
class MainControl {
  @FXML private var myButton: Button = null

  @FXML def buttonAction(event: ActionEvent) {
    System.exit(0)
  }
}
```

5.2 TextField

This allows a user to enter a single line of unformatted text.

A TextField can be used for several purposes, as seen in the docs. One function that you might find particularly useful is the `getText()` function. After a user writes text into a TextField, the TextField contains that text in `String` format. For this example, we will show you how you can get the text when a user clicks a button, and print that `String` to the console.

```
class MainControl {
  @FXML private var printButton: Button = null
  @FXML private var myTextfield: TextField = null

  @FXML def printAction(event: ActionEvent) {
    var input: String = myTextfield.getText()
    println(input)
  }
}
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/cs018/feedback`.