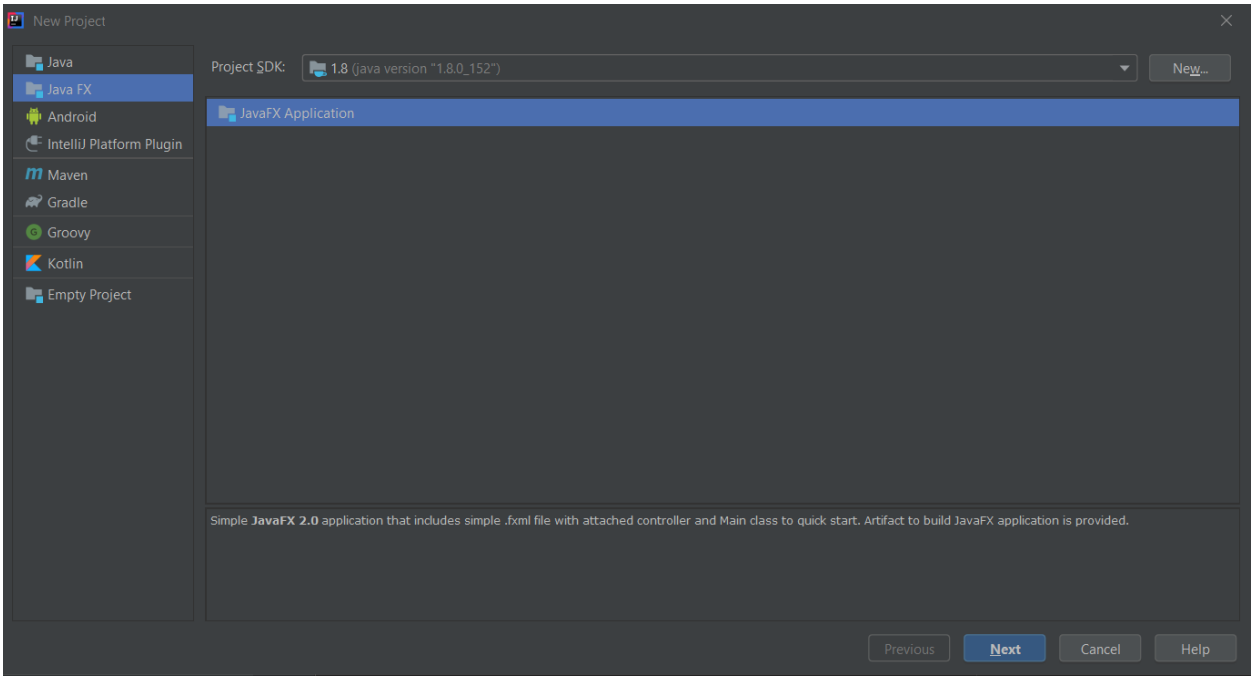
CS 2340- FXML and JavaFX (Java 8)

Introduction

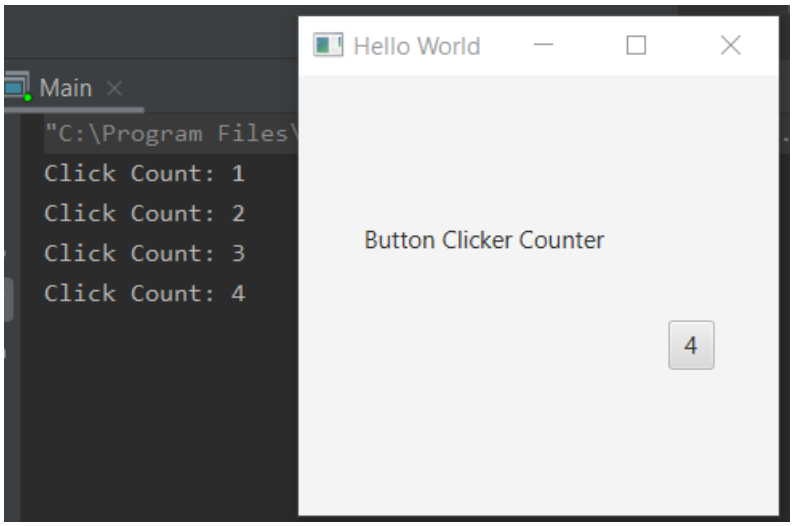
If your only experience with JavaFX so far was in CS 1331, you might be thinking that writing an entire project to use JavaFX GUIs would end up in an extremely cluttered Application file. In fact, you can separate out the stylistic UI factors from their programming logic across multiple files by making use of FXML. FXML is an XML-based markup language for representing Java objects graphically and works well with JavaFX for organizing and styling GUIs. This guide will serve to help you get started with adding FXML to your project, and it will go over a variety of options and features you can make use of by scripting in FXML. At the **end of this guide** are several **helpful links** to Oracle's documentation for FXML/CSS and their official tutorials. An alternative to scripting in FXML is making use of SceneBuilder, which is a visual layout tool for designing FXML without coding in FXML and comes built-in to JavaFX projects on IntelliJ.

If you're unsure of how to start a JavaFX project in IntelliJ, don't worry. Click on "File>New Project" and then select "Java FX" on the left-hand side of the New Project Menu:



Getting Started with FXML

This section will show you how to make a basic GUI that counts clicks of a button and prints the count to a console in order to demonstrate the basic attributes of using FXML.



1. Loading FXML Files *(check Appendix 1 for sample file)*

If you are using IntelliJ to start your JavaFX project, you might have noticed three files show up under the "sample" package IntelliJ generated for you: main.java, sample.fxml, and controller.java (sample files located in Appendices I, II, and III respectively). JavaFX projects will generally feature a similar cast of files: a java file dedicated to an application or scene, an FXML file for format, and a java controller file that "injects" fields and values into the format generated by the FXML.

To make a Scene in JavaFX using FXML, add two new files: a FXML file and a .java file (this is going to be the controller for your FXML file). Then, make sure to add a line of code to your application that loads the FXML file, commonly done by calling the load() method of the FXMLLoader class using the path to your FXML file. An example of loading FXML into a Scene for a Stage is shown below (also located in Appendix I):

```
Parent root = FXMLLoader.load(getClass().getResource("sampleFXML"));
primaryStage.setTitle("Hello World");
primaryStage.setScene(new Scene(root, 300, 275));
primaryStage.show();
```

Your program will now be using the contents of the FXML file and the controller file to generate this scene's graphical display.

2. Setting up the FXML file *(check Appendix II for sample file)*

Over in the FXML file, you can start scripting the layout, adding panes, labels, and various other JavaFX Nodes. If you are familiar with HTML, you might notice some similarities with FXML, both being XML-based in nature. Below is an example FXML file (also located in Appendix II):

```
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<GridPane fx:controller="sample.Controller"
          xmlns:fx="http://javafx.com/FXML" alignment="center" hgap="10"
vgap="10">
    <children>
        <Label text="Button Clicker Counter">
            <GridPane.rowIndex>0</GridPane.rowIndex>
            <GridPane.columnIndex>0</GridPane.columnIndex>
        </Label>
    </children>
</GridPane>
```

Note that now instead of importing javafx.scene elements into your Application file, you're importing them directly into your FXML file. In the example above, you can see the hierarchy of Nodes in the scene that this FXML file represents. The "first" node in the example is a GridPane, which is the root element of this FXML file. It is responsible for both listing the FXML controller attribute "fx:controller", which links the FXML to the corresponding controller, and the "xmlns:fx" attribute, which is a requirement for the FXML to function properly.

The root element can have "children" that fill out the GUI. In the example above, the GridPane root has two child nodes: a Label and a Button. Nodes like Label and Button have several attributes that can be set in the FXML file or in its controller file. Adding a node to an FXML file is as simple as adding a Node-type element like <Button> to the file as can be seen on the next page.

```
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<GridPane fx:controller="sample.Controller"
          xmlns:fx="http://javafx.com/FXML" alignment="center" hgap="10"
vgap="10">
    <children>
        <Label text="Button Clicker Counter">
            <GridPane.rowIndex>0</GridPane.rowIndex>
            <GridPane.columnIndex>0</GridPane.columnIndex>
        </Label>
        <!-- added a Button node here -->
        <Button fx:id="counterClicker" onAction="#showCount">
            <GridPane.rowIndex>4</GridPane.rowIndex>
            <GridPane.columnIndex>4</GridPane.columnIndex>
        </Button>
    </children>
</GridPane>
```

In the example above, the Label's text is set in the FXML file, but the Button's text is not. You will notice that the button has two attributes: "fx:id" and "onAction." This example uses a controller file to set the Buttons text and behavior, which we will cover in the next section.

3. FXML Controller Classes *(check Appendix III for sample file)*

An FXML controller class is a Java file that works with an FXML file to display a GUI. With a controller class, you can implement background logic for "tagged" parts in the FXML file. In the FXML file, you can tag a node as being represented in the controller file by adding the attribute "fx:id" to the node. Once you have done that in the FXML file, over in the controller file you will need to set up a node of the same type and name as listed in the FXML file.

Ex.

In the FXML file:

```
<Button fx:id="counterClicker" onAction="#showCount">
    <GridPane.rowIndex>4</GridPane.rowIndex>
    <GridPane.columnIndex>4</GridPane.columnIndex>
</Button>
```

In the controller file:

```
@FXML
private Button counterClicker;
```

Note the @FXML tag placed in the controller file before the Button counterClicker was declared. This tag allows the FXML file to use private and protected elements like counterClicker in the controller file when it is generating the visuals of the GUI. A Node object tagged in the controller file can now be dynamically changed in with JavaFX instead of remaining static in an FXML file.

For any tagged element that is implemented in the controller file, you can use the initialize method to manipulate their values on startup.

```
private static int counter = 0;


@FXML
public void initialize() {
    counterClicker.setText("" + counter);
}
```

The initialize method is called after @FXML tagged objects are registered, so the method can manipulate those objects attributes to make up their "initial", visible state. Notice that the controller file can have its own fields (like the counter variable) that aren't directly shown in the FXML. However, what if we want to change how things look after the FXML has been rendered and initialized?

Remember that "onAction" attribute we had placed on the Button in the FXML file?

```
<Button fx:id="counterClicker" onAction="#showCount">
```

In the FXML file, the Button having the "onAction" attribute means that any time the button (counterClicker) registers an action event, it will call the showCount method in the controller file. ActionEvent handlers like our showCount method can have a wide variety of behaviors like checking for values across your system, showing/hiding nodes, moving between scenes, or, in our case, changing the text field of our button as shown below.

```
@FXML
private void showCount(ActionEvent event) {
    counter++;
    System.out.println("Click Count: " + counter);
    counterClicker.setText("" + counter);
}
```

By now, we should have a basic familiarity with FXML and be able to add the necessary FXML and controller files to our code.

Custom FXML Components *(check Appendices IV and V for sample files)*

This section draws on concepts discussed in Oracle's Introduction to FXML Custom Components Section.

FXML files can be filled out with layouts, controls (like buttons), charts, and even more types of nodes. You can also make your own custom FXML components for use in a FXML file, which can improve a project/system with FXML reusability and ease of editing for once-repeating sections of FXML code. There are a variety of different ways that you can do so. If you want to write an extension of an existing node, you can write a Java file extending a node (ex. CustomButton extends Button) and import it at the beginning of your FXML file. For a more in-depth customized component (maybe incorporating more than one node), we will need a component-defining FXML file and a java file that both extends a node (perhaps such as a Region or Pane) and acts as a controller for that defining FXML file.

An example of the custom component's FXML file is shown as below:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>


<fx:root    type="javafx.scene.layout.HBox"
            xmlns="http://javafx.com/javafx"
            xmlns:fx="http://javafx.com/fxml">
    <TextField fx:id="textField"/>
    <Button text="Click to Print" onAction="#printText"/>
</fx:root>
```

Note that instead of starting the FXML file with a layout we start it the <fx:root> element and that we do not have the "fx:controller" attribute. Also note that IntelliJ will mark "onAction's" as incorrect in the absence of an "fx:controller" attribute in the FXML file's root element. When using the <fx:root> element, you should fill out the basic type of (typically) layout. This type is what your implementing controller file should extend. An example of the controller file is in Appendix V.

If you are confused about the StringProperty objects, don't worry. Later in the course, we will be looking into the Observer design pattern and attempting to have objects that contain other objects in order to register change.

The controller/root implementor file in the page above is similar to the controller file seen before in this guide with the main difference being the introduction of a controller. In this case, the CustomComponent.java's constructor method ensures that the FXML file recognizes it as both the root and controller for the component. This allows your new custom FXML component to have its own distinct controller object for each of its instantiations in the FXML file. It also

allows the component to have its own controller despite not always being the root node of a scene it is added to. The custom component controller is located in Appendix V.

For reference, below is an FXML file for a scene that makes use of our custom component:

```
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import sample.CustomComponent?>
<GridPane fx:controller="sample.Controller"
          xmlns:fx="http://javafx.com/fxml" alignment="center"
          hgap="10" vgap="10">
    <children>
        <Label text="Button Clicker Counter">
            <GridPane.rowIndex>0</GridPane.rowIndex>
            <GridPane.columnIndex>2</GridPane.columnIndex>
        </Label>
        <Button fx:id="counterClicker" onAction="#showCount">
            <GridPane.rowIndex>3</GridPane.rowIndex>
            <GridPane.columnIndex>2</GridPane.columnIndex>
        </Button>
        <CustomComponent>
            <GridPane.rowIndex>6</GridPane.rowIndex>
            <GridPane.columnIndex>2</GridPane.columnIndex>
        </CustomComponent>
        <CustomComponent>
            <GridPane.rowIndex>5</GridPane.rowIndex>
            <GridPane.columnIndex>2</GridPane.columnIndex>
        </CustomComponent>
    </children>
</GridPane>
```

We have now constructed and made use of our own custom FXML component! The next section of this guide will briefly discuss various features of FXML and the use of SceneBuilder.

Other Topics in FXML

1. Should I use FXML or SceneBuilder?

As Oracle says at the beginning of their FXML guide: "*Just as some developers prefer to work directly in the XML code, other developers prefer to use a tool to author their XML. The same is true with FXML.*" SceneBuilder is a visual tool where you can add nodes, customize them, and stylize them for JavaFX applications. Also, SceneBuilder is generating FXML code while you work in it, allowing you to use it to check how existing FXML code looks as a JavaFX Scene or to build off of SceneBuilder-made FXML by fine-tuning the code yourself. Be careful when switching between SceneBuilder and FXML though, as SceneBuilder can quickly generate a lot of unfamiliar FXML attributes, tags, and style-components that might elongate an FXML file and make the FXML code itself hard to read or modify.

2. FXML and CSS

If you have prior experience or want to go deeper in stylization in JavaFX, you can use JavaFX's version of CSS. Nodes can be styled inline using the setStyle method, or a Scene can set a CSS stylesheet that it uses given the stylesheet's path. Using a CSS stylesheet allows you to style type's of nodes, specify nodes to style with an id, and even quickly skin the style of a scene by working with the .root of a scene. If you want to familiarize yourself with CSS in JavaFX, I highly suggest looking over Oracle's CSS links in the Resource section of this guide. Also attached to this guide in the appendix is the Modena stylesheet, JavaFX's "default" stylesheet.

3. Scripting in FXML

It is possible to embed or import any JVM language (including JavaScript) into an FXML file by using the <fx:script> tag. Oracle's example of an embedded script is as follows:

```
<?language javascript?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns:fx="http://javafx.com/fxml">
    <fx:script>

    function handleButtonAction(event) {
       java.lang.System.out.println('You clicked me!');
    }
    </fx:script>

    <children>
        <Button text="Click Me!" onAction="handleButtonAction(event);"/>
    </children>
</VBox>
```

Note the <?language javascript?> at the top of the example and the JavaScript code between the <fx:script> tags.

Alternatively, you may use the <fx:script> tag to import code such as from a JavaScript file in order to separate FXML code from handler code by giving the tag the source attribute, as in:

FXML File Ex. (from Oracle)

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns:fx="http://javafx.com/fxml">
    <fx:script source="example.js" charset="cp1252"/>

    <children>
        <Button text="Click Me!" onAction="handleButtonAction(event);"/>
    </children>
</VBox>
```

JavaScript File Ex. (from Oracle)

```
function handleButtonAction(event) {
   java.lang.System.out.println('You clicked me!');
}
```

Note that you no longer need to include the <?language javascript?> in the FXML file once you have separated out the JavaScript into an external file.

Conclusion

Hopefully, this guide has served to familiarize you with the basics of using FXML in a JavaFX application and point you toward several helpful resources for expanding your knowledge and applied skills for formatting and stylizing JavaFX applications. When used effectively for this course, use of FXML (or Scenebuilder) can allow you to efficiently apply your designs to the "realities" of Java and JavaFX programming for your project. Be sure to make use of the resource links in the next page, as they lay the path for more in-depth use of FXML and JavaFX. Oracle's Introduction to FXML is extremely useful for examining the different applications of FXML code that you may be unfamiliar with. If you plan to apply CSS to your project, the CSS reference guide is essential for getting to know how to use it in JavaFX. Good luck moving forward with your JavaFX project, and I hope you make use of the FXML basics laid out in this guide!

Resource Links:

Oracle Documentation:

JavaFX/Scene Builder 2 Guides Page: https://docs.oracle.com/javase/8/javase-clienttechnologies.htm
(This link goes to Oracles page for its collection of tutorials and reference guides for JavaFX)

JavaFX API: https://docs.oracle.com/javase/8/javafx/api/toc.htm

Getting Started With JavaFX: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/get_start_apps.htm#JFXST804

FXML Helpful Links

Introduction to FXML: https://docs.oracle.com/javase/8/javafx/api/javafx/FXML/doc-files/introduction_to_FXML.html

JavaFX: Mastering FXML: https://docs.oracle.com/javase/8/javafx/FXML-tutorial/preface.htm

Working With Layouts in JavaFX: https://docs.oracle.com/javase/8/javafx/layout-tutorial/preface.htm

Properties and Bindings: https://docs.oracle.com/javase/8/javafx/properties-binding-tutorial/binding.htm#JFXBD107

CSS Helpful Links

JavaFX CSS Reference Guide: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html

Skinning JavaFX Applications with CSS: https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/css_tutorial.htm#JFXUI733

Fancy Forms with CSS: https://docs.oracle.com/javase/8/javafx/get-started-tutorial/css.htm

Stack Overflow:

Basic Overview of FXML and FXML Controllers:
https://stackoverflow.com/questions/33881046/how-to-connect-fx-controller-with-main-app#:~:text=Within%20each%20pair%2C%20the%20FXML,processes%20user%20input%2C%20etc).&text=Loads%20the%20FXML%20file,calling%20its%20no%2Dargument%20constructor

Loading new FXML in the same scene: https://stackoverflow.com/questions/18619394/loading-new-fxml-in-the-same-scene
(Check out the answer detailing Frameworks)

Appendix I: Sample JavaFX Main File

```
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception{
        Parent root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        primaryStage.setTitle("Hello World");
        primaryStage.setScene(new Scene(root, 300, 275));
        primaryStage.show();
    }


    public static void main(String[] args) {
        launch(args);
    }
}
```

Appendix II: Sample FXML File

```
<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import sample.CustomComponent?>
<GridPane fx:controller="sample.Controller"
          xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
    <children>
        <!-- child nodes go here, ex Label node below -->
        <Label text="Button Clicker Counter">
            <!-- GridPane's require row and column index to be specified -->
            <GridPane.rowIndex>0</GridPane.rowIndex>
            <GridPane.columnIndex>2</GridPane.columnIndex>
        </Label>
    </children>
</GridPane>
```

Appendix III: Sample Controller File

```
package sample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Button;

public class Controller {
    private static int counter = 0;

    @FXML
    private Button counterClicker;

    @FXML
    private void showCount(ActionEvent event) {
        counter++;
        System.out.println("Click Count: " + counter);
        counterClicker.setText("" + counter);
    }

    @FXML
    private void initialize() {
        counterClicker.setText("" + counter);
    }
}
```

Appendix IV: Sample Custom Component FXML File

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>


<fx:root    type="javafx.scene.layout.HBox"
            xmlns="http://javafx.com/javafx"
            xmlns:fx="http://javafx.com/fxml">
    <TextField fx:id="textField"/>
    <Button text="Click to Print" onAction="#printText"/>
</fx:root>
```

Appendix V: Sample Custom Component Controller

```
package sample;

import javafx.beans.property.StringProperty;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;

import java.io.IOException;

public class CustomComponent extends HBox {
    @FXML
    private TextField textField;

    private static int componentCount = 0;
    private int countID;

    public CustomComponent() {
        countID = componentCount;
        componentCount++;
        FXMLLoader fxmlLoader = new
                FXMLLoader(getClass().getResource(
                        "CustomComponent.fxml"));
        fxmlLoader.setRoot(this);
        fxmlLoader.setController(this);

        try {
            fxmlLoader.load();
        } catch (IOException exception) {
            throw new RuntimeException(exception);
        }
    }

    public String getText() {
        return textProperty().get();
    }

    public void setText(String value) {
        textProperty().set(value);
    }

    public StringProperty textProperty() {
        return textField.textProperty();
    }

    @FXML
    protected void printText() {
        System.out.println("ID " + countID + " posts: " + getText());
        setText("");
    }
}
```