Introduction to JavaFX for Beginner Programmers

Robert Ball, Ph.D.

August 16, 2017
Version 0.1.4

© 2017
All rights reserved.

This work may be distributed or shared at no cost, but may not be modified.

Contents

1 Introduction to JavaFX **1**

1.1 Introduction 2

1.2 Two Approaches 4

1.2.1 Approach 1 - Hand-Written Code 4

1.2.2 Approach 2 - Drag And Drop 4

1.2.3 Layouts 6

1.2.4 Incorporating Java code with JavaFX 17

2 Common JavaFX controls and their most common methods **25**

3 Common JavaFX Errors and Their Fixes **27**

4 Graphics Introduction - How to Draw with Javafx **29**

4.1 Understanding the Origin 29

4.2 Approach 1 - Use Pre-Created Shapes 31

4.3 Approach 2 - Draw Your Own Shapes 31

5 How to make an executable Java program **33**

5.0.1 Executable JAR file in JGrasp 33

Chapter 1

Introduction to JavaFX

What is JavaFX?

JavaFX is the latest GUI (Graphical User Interface) environment that Java uses. Its predecessors include AWT and Swing. **Swing** is a GUI toolkit that made creating GUI's with Java much easier. It is still used heavily in today's world, but is no longer being actively developed.

According to Oracle, JavaFX is "a set of graphics and media packages that enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms." In other words, JavaFX is the latest way to create GUI applications with Java.

JavaFX was created and is maintained by Oracle, but you can use it with other languages as well, like JRuby, Scala, Jython (a of Python), Groovy, and JavaScript! In addition, you can use the web language CSS (Cascading Style Sheets) to alter the appearance of your GUI application without changing your code.

Although you can use many different languages with JavaFX, we will focus on how to use it with Java in this book.

The official documentation for JavaFX is the following url: `http://docs.oracle.com/javase/8/javafx/api/toc.htm`

1.1 Introduction

Let's get going. The following code is a very simple JavaFX application.

Notice three main things. First, that the class uses "extends Application." What this does is tell Java that you are going to use inheritance to make this application use JavaFX.

Next it has a required method named "start(Stage primaryStage)." The "primaryStage" is what is going to appear. Think of the "Stage" as a stage that actors stand on for theater productions.

Finally, we tell the Stage to appear by "primaryStage.show();" This is what makes the application visible. See figure 1.1 to see what it looks like when the program is run on a Mac.

```
import javafx.application.Application;
import javafx.stage.Stage;

public class Ch1_1 extends Application {
    public void start(Stage primaryStage) {
        primaryStage.show();
    }

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

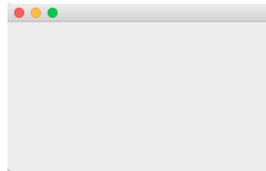

Figure 1.1: What the class Ch1_1 looks like when run on a Mac.

Let us make the application a littler more interesting. After all, an empty application is very boring and useless.

The following code is slightly different from the one above. First, it has a label. The label is just text that will appear on the screen. Second it has a "StackPane" layout - which means that the the text will appear in the middle of the application. (See section 1.2.3 for

more information on "StackPane" and layouts in general.) The next line adds the label to the StackPane. The StackPane has to then be added to a "Scene."

A "Scene" is similar to scenes in theater where the scenes can be changed, but the stage is always the same. For example, in theater, you might have an opening scene with a blue backdrop and a couch. Then, when the scene changes the backdrop becomes red and a chair replaces the couch. Regardless of how many scenes are in a play at the theater, there is only one stage where the actors always perform.

When the "Scene" is all configured then it is added to the Stage. Figure 1.2 shows what it looks like on a Mac.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class Ch1_2 extends Application {
   public void start(Stage primaryStage) {

      Label label1 = new Label("I love JavaFX!"); //show text

      StackPane root = new StackPane(); //create a layout
      root.getChildren().add(label1); //add the Label to the
         layout
      Scene scene = new Scene(root,100,100); //add the StackPane
         to the scene and set's the width to be 100 pixels and
         the height to be 100 pixels
      primaryStage.setScene(scene); //set the Scene

      primaryStage.show(); //show the Stage
   }

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

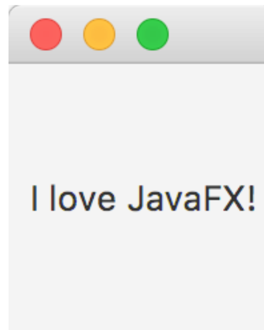

Figure 1.2: What the class Ch1_2 looks like when run on a Mac.

1.2 Two Approaches

There are two main approaches to creating Java GUI's. You can either write all the code by hand or you can use a drag-and-drop application.

1.2.1 Approach 1 - Hand-Written Code

The first approach is to create the GUI completely by hand-written code. This approach is well worth your time. You will have complete control over every aspect of the what happens. This is what I have shown above.

However, this approach requires months of digging into documentation and is not the approach we will take with this book. This used to be the only way to create GUI's in the long forgotten past.

1.2.2 Approach 2 - Drag And Drop

The second approach is to create the GUI using a drag-and-drop application, like Scene Builder (`http://gluonhq.com/open-source/scene-builder/`).

Although you can use Scene Builder with Eclipse or NetBeans, both of those IDE's are beyond the scope of this book. If you are advanced enough to be using Eclipse or NetBeans then you are advanced enough to figure out how to incorporate Scene Builder into those IDE's on your own. Here is a link for Scene Builder with NetBeans: `http://docs.oracle.com/javafx/scenebuilder/1/use_java_ides/sb-with-nb.htm`.

The approach that this book is going to take is to design the GUI in Scene Builder and write the code in JGrasp. The reason for this approach is that it allows the beginning student to see exactly how the GUI is put together with no magic, but makes it easier by having to only drag-and-drop the GUI components.

I presume at this point that you already have a very fundamental knowledge of how Java works and that you have both Java and JGrasp installed. The next step is to install Scene Builder. With a web browser, navigate to `http://gluonhq.com/open-source/scene-builder/`. Scroll down to the bottom of the page and download the executable jar (see figure 1.3). I will be using Scene Builder version 8.2.0 for this book.

Download Scene Builder

The latest version of Scene Builder is **8.2.0**, it was released on **May 18, 2016**.

To be kept informed of Scene Builder releases, consider subscribing to the Gluon Newsletter.

Product	Platform	Download
Scene Builder	Executable Jar	Download
Scene Builder	Windows Installer (x86) info	Download
Scene Builder	Windows Installer (x64) info	Download
Scene Builder	Mac OS X dmg	Download
Scene Builder	Linux RPM (64-bit)	Download
Scene Builder	Linux Deb (64-bit)	Download
Scene Builder	Linux RPM (32-bit)	Download
Scene Builder	Linux Deb (32-bit)	Download
Scene Builder Kit info	Jar File	Download

Figure 1.3: Download the Executable Jar (circled in red).

Double-click on the executable jar to start it. Figure 1.4 shows what Scene Builder looks like when it first starts up.

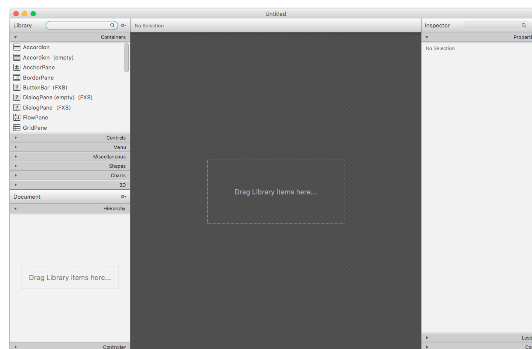

Figure 1.4: Scene Builder at start up.

In order to use Scene Builder you need to first understand how layouts work.

1.2.3 Layouts

Layouts are how GUI's are put together. The layout is how GUI controls are placed on the screen. A GUI **control**, also known as a GUI **widget** is an interactive element in the GUI, like a button, scrollbar, or textfield.

There are many different layouts. The following are some of the more common layouts:

- **BorderPane** - A basic layout that gives you five areas: top, left, right, bottom, and center.
- **HBox** - A basic layout that orders the GUI controls in a horizontal (thus the "H") line.
- **VBox** - A basic layout that orders the GUI controls in a vertical (thus the "H") line.
- **GridPane** - A basic layout that orders the GUI controls in a grid. For example, a grid might be 2 row by 2 columns.
- **StackPane** - A basic layout that put all the GUI controls in a stack, in other words, right on top of each other.

Let us start with the BorderPane. With Scene Builder open, make sure that you can see the different "Containers" - the top left part of Scene Builder. See Figure 1.5.

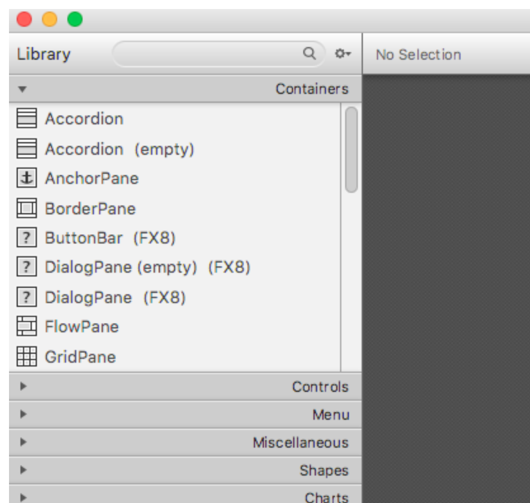

Figure 1.5: Scene Builder containers.

Find “BorderPane” - the fourth from the top, click on it, then drag it to the middle of the application, where is has the text “Drag Library items here...” Figure 1.6 shows what Scene Builder should now look like.

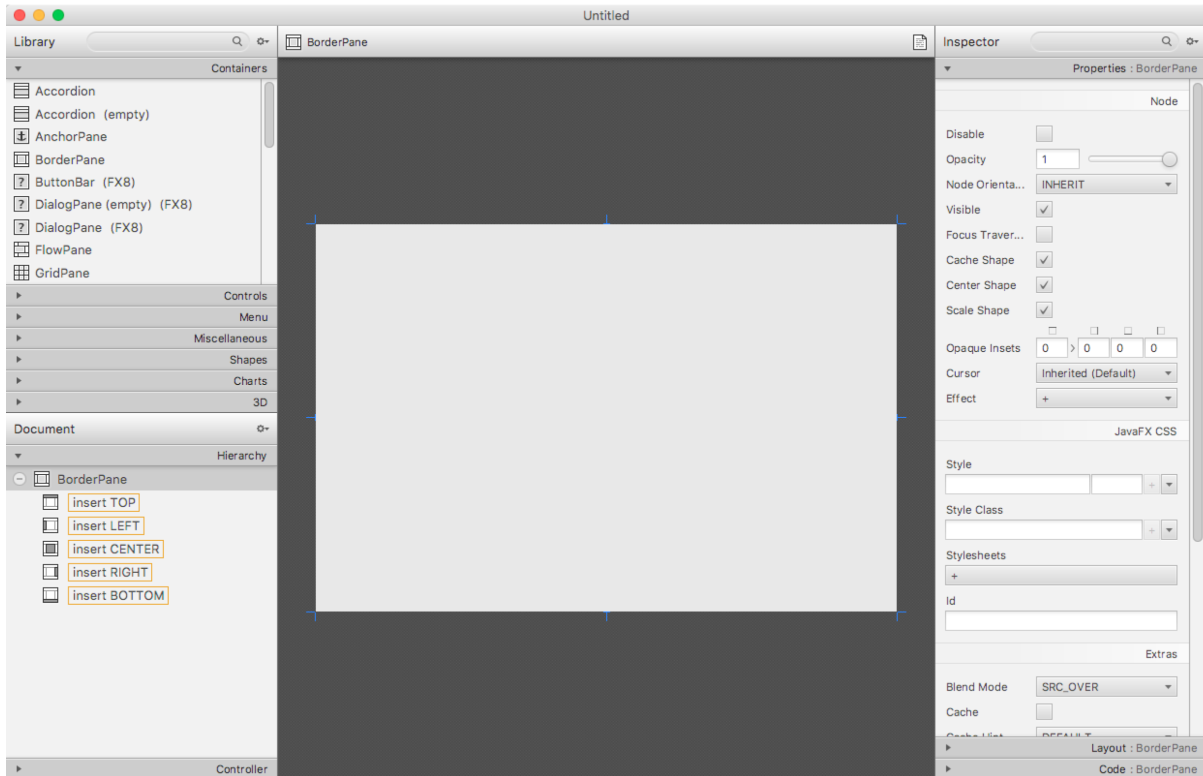

Figure 1.6: Scene Builder with the Border Layout.

Now, let’s do something interesting. Let’s add a label, a textfield, and a button. First, click on “Controls.” “Controls” is on the left, right below “Containers.” The first control that you should see is “Button.” Click on “Button” then drag it to the middle of your layout. Figure 1.7 shows what this looks like. Notice that when you drag a control over a BorderPane layout in Scene Builder that it lets you see which area (top, left, right, bottom, or center) you are putting the control.

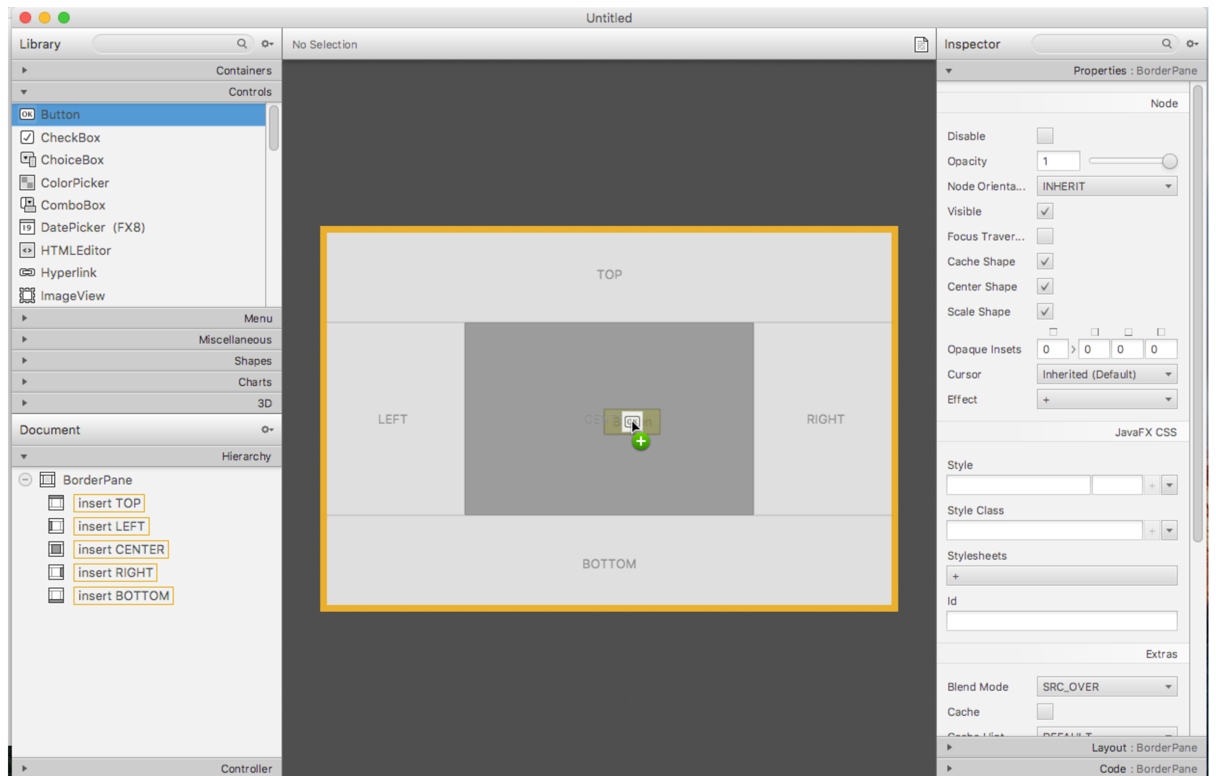

Figure 1.7: Placing a Button in the center of a BorderPane layout in Scene Builder.

The button should be highlighted, be in the center of the BorderPane and have the text, “Button.” Let’s change the text so that it now says, “Multiply” instead of “Button.”

If the button is not highlighted then click on it. In the upper-right corner of Scene Builder you will see “Properties.” If “Properties” is closed then click on it to open it. Find the text “Button” and change it to “Multiply.” Let’s change the color in “Text Fill’ to another color. Any color you want is fine - I will use red. Let’s change the size of the font too. I chose to change it to 18px. You can change the font by clicking on the drop down box called “Font.” See Figure 1.8 for more details.

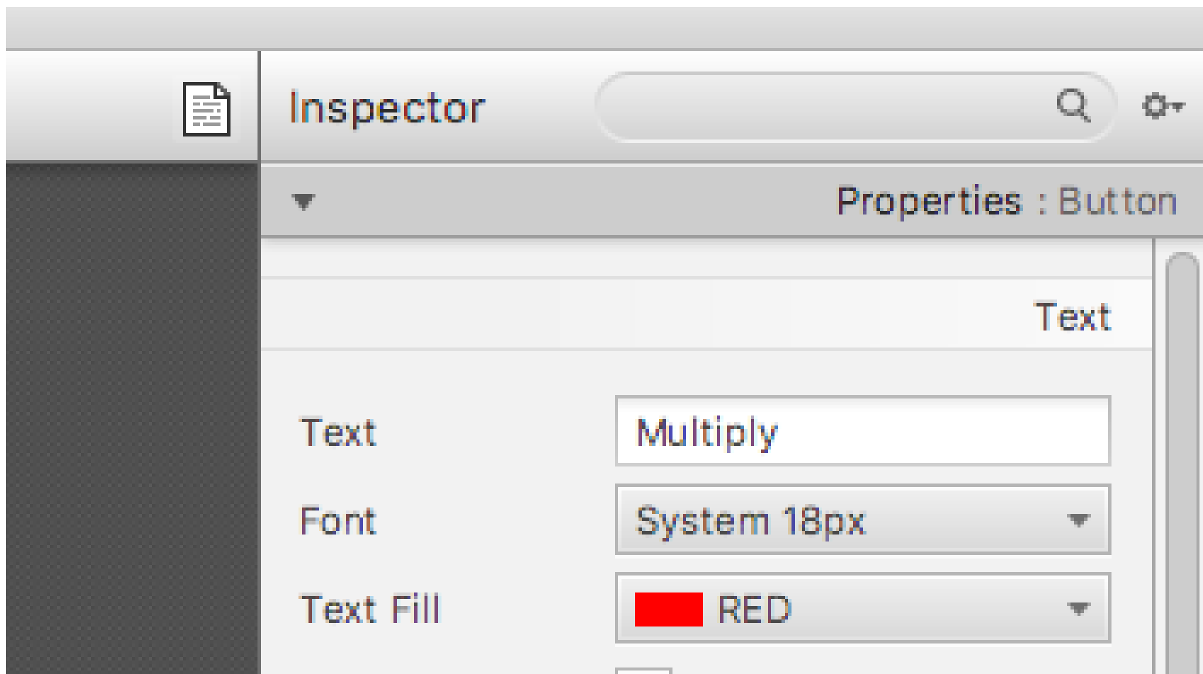

Figure 1.8: Showing the changed properties of the Button.

Now let's add another layout on top of the BorderPane. In the top-left of Scene Builder click on "Containers" again. This shows the layouts again. Scroll down until you see the "HBox" layout. Click on it then drag it to the "top" part of the already place BorderPane. See figure 1.9 for details.

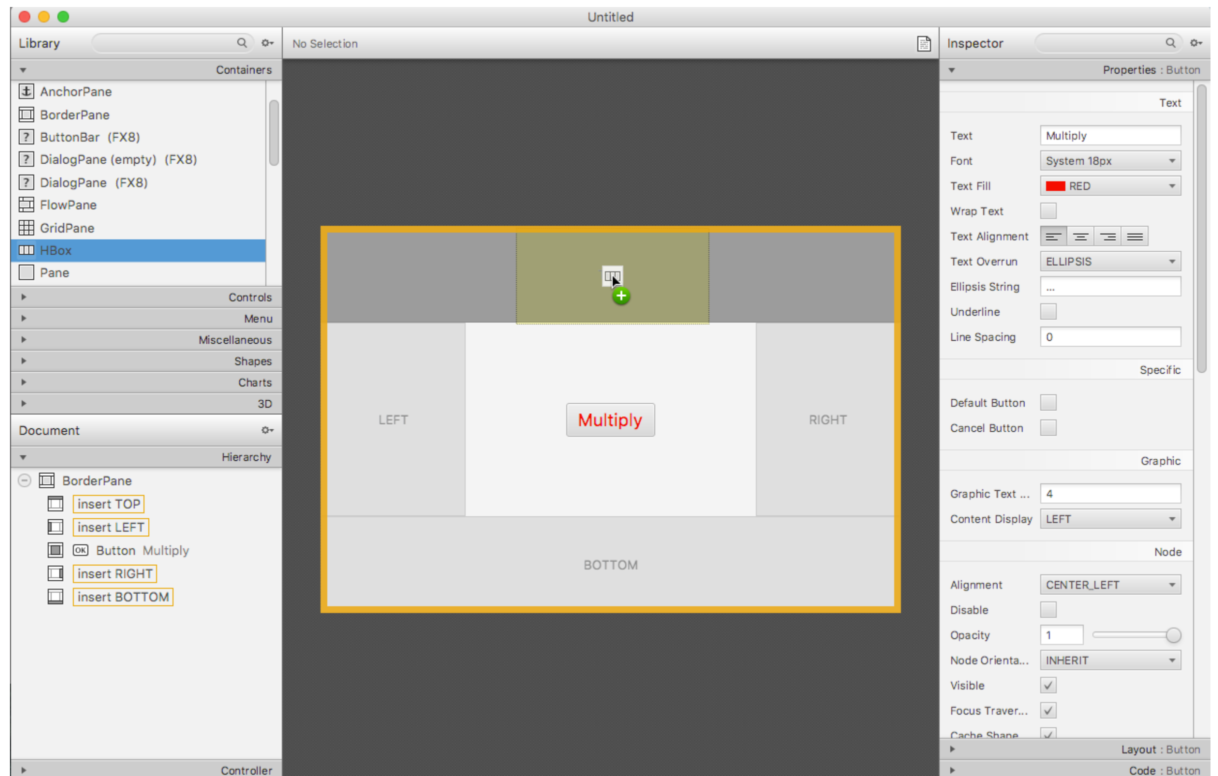

Figure 1.9: Placing an HBox layout in the top of a BorderPane layout in Scene Builder.

Now let's add our label. Click on "Controls" again, right below "Containers." This will show "Button" at the top of the list. Scroll down until you see "Label." Click on "Label" then drag it onto the HBox. See figure 1.10 for details.

Let's change the Label to be more interesting. With the Label highlighted I change the properites - upper-right corner. I changed my text property to be "Multiply by 5:", I changed the font to be 18px, and I changed the color.

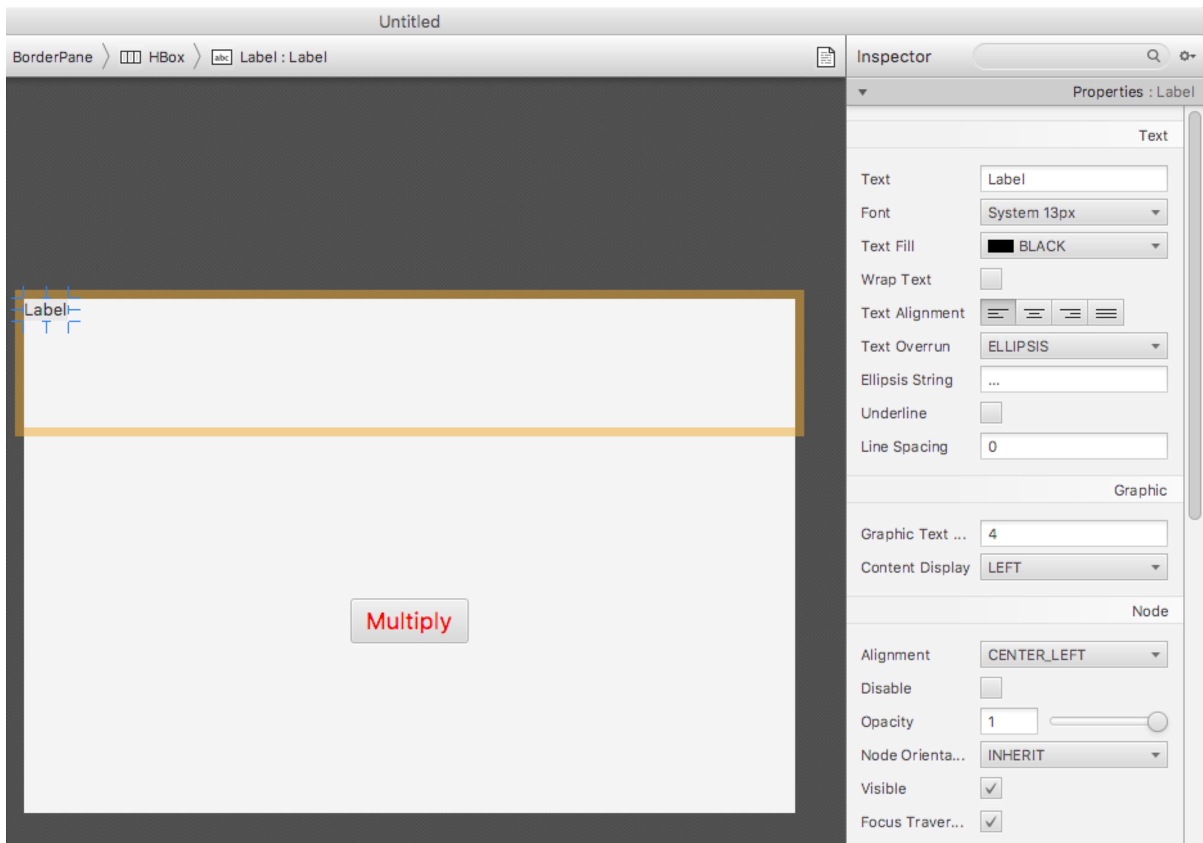

Figure 1.10: Placed a Label in the HBox layout.

In the same way as you added the label, add a "TextField." Under "Controls" - on the left - scroll down until you see "TextField" - it is sixth from the bottom. Click on it then drag it to the right of the Label. Scene Builder should now look something like figure 1.11.

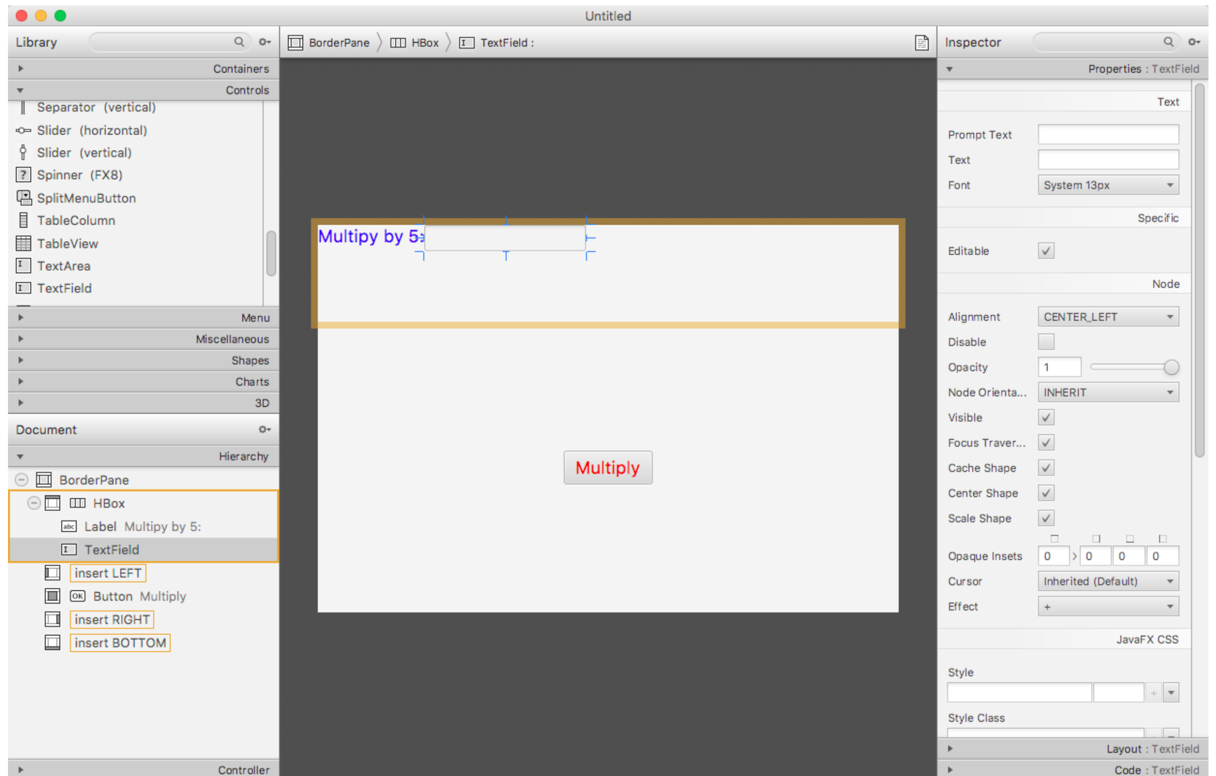

Figure 1.11: Placed a TextField to the right of the Label in the HBox layout.

Before we finish there are two things we need to do:

- Give the Button an id so that we can process the button click.
- Give the TextField an id so that we can get input from it.

In order to give the button an id, we need to click on it. The properties will now show the properties of the button. On the bottom of the properties list this is a section called "Code." Specifically, it should look like "Code : Button." Click on "Code" and you will be provided with the opportunity to give it an "fx:id." In the "fx:id" TextField, enter the text, "myButton." It should look like figure 1.12.

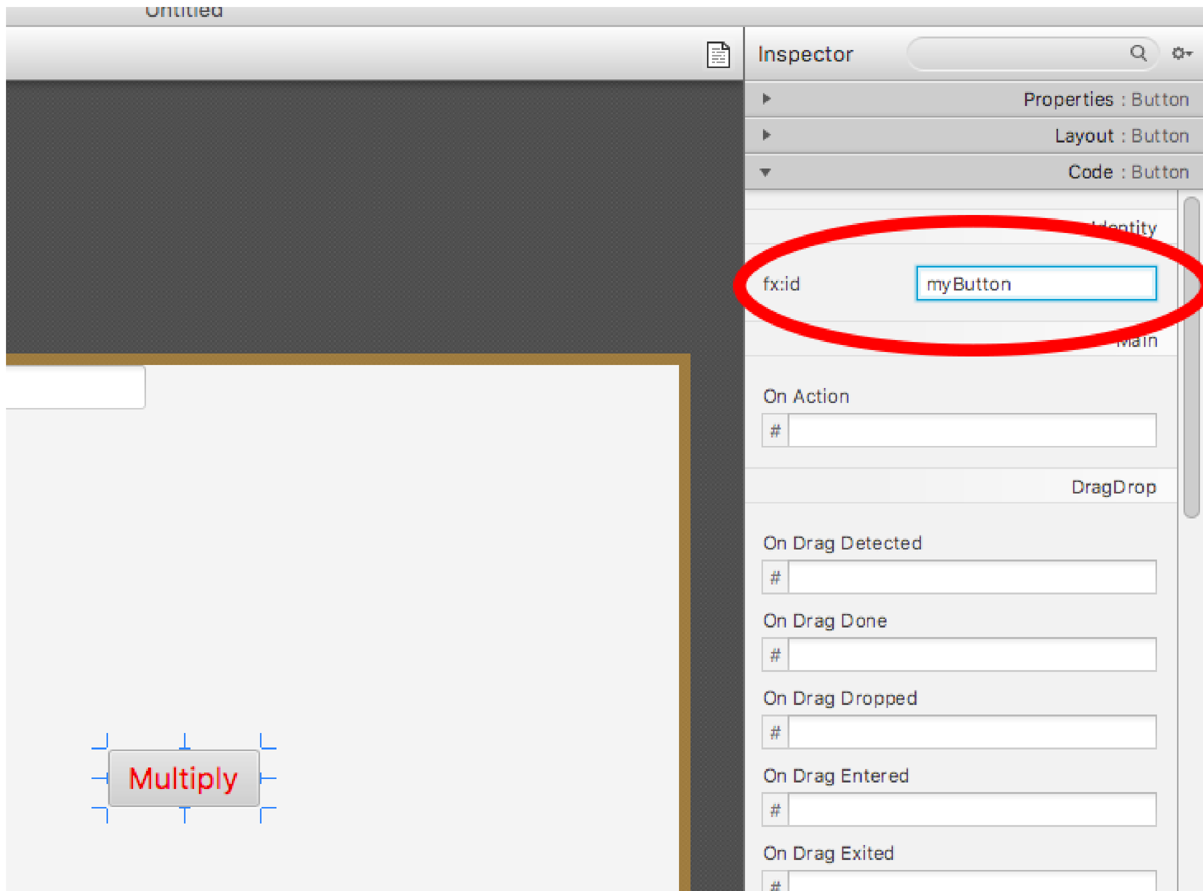

Figure 1.12: Showing the fx:id of myButton for the highlighted button.

We will also need to take care of any button clicks that the buttons will get when the user clicks on it. In the "On Action" field, add "handleButton" - see figure 1.13.

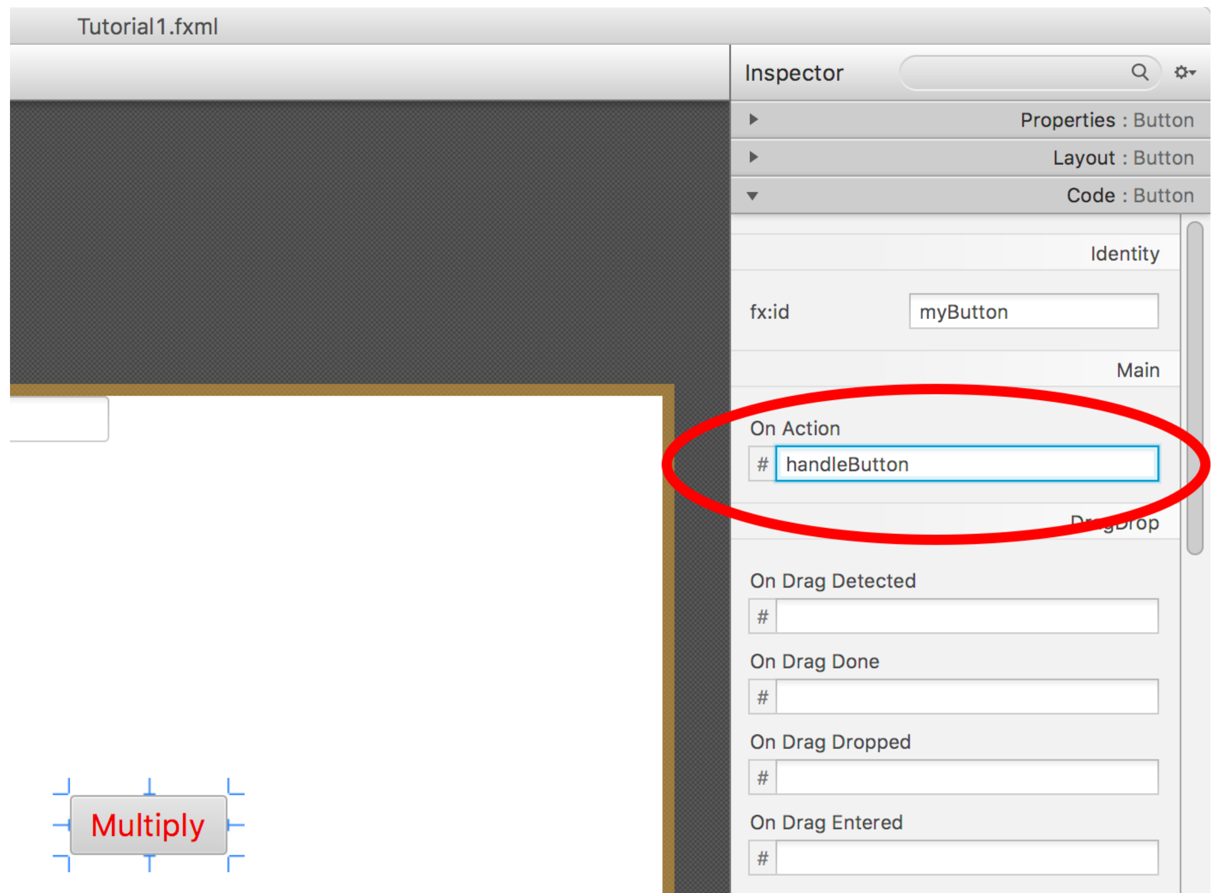

Figure 1.13: Showing the addition of "handleButton" to handle the button click on the highlighted button.

Now give TextField and id - call it "myTextField." To do it, click on the TextField, go the code area of the properties, and enter "myTextField" in the "fx:id."

There is no need to give the TextField an "On Action" method - just leave it blank for the TextField.

Now what? Save it! I am going to save our GUI as "Tutorial1." Find the menu, click "File" then "Save as..." then "Tutorial1.fxml."

If you were to open up "Tutorial1.fxml" - which you do not have to do! - then you will see something similar to the following code that was generated for you:

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

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.BorderPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.text.Font?>

<BorderPane maxHeight="-Infinity" maxWidth="-Infinity"
    minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0"
    prefWidth="600.0" xmlns="http://javafx.com/javafx/8.0.91"
    xmlns:fx="http://javafx.com/fxml/1">
   <center>
      <Button fx:id="myButton" mnemonicParsing="false"
          onAction="#handleButton" text="Multiply" textFill="RED"
          BorderPane.alignment="CENTER">
         <font>
            <Font size="18.0" />
         </font>
      </Button>
   </center>
   <top>
      <HBox prefHeight="100.0" prefWidth="200.0"
          BorderPane.alignment="CENTER">
         <children>
            <Label text="Multipy by 5:" textFill="#3c00ff">
               <font>
                  <Font size="18.0" />
               </font>
            </Label>
            <TextField fx:id="myTextField" />
         </children>
      </HBox>
   </top>
</BorderPane>
```

The above code is what Scene Builder generated for you so that you do not have to hand write the GUI yourself.

If you are curious, the following is what the hand-written code looks like:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class HandWrittenCode extends Application {

    public void start(Stage primaryStage) {

        HBox hbox1 = new HBox();

        Label label1 = new Label("Multipy by 5:"); //show text

        TextField myTextField = new TextField();

        hbox1.getChildren().addAll(label1,myTextField);

        BorderPane borderPane1 = new BorderPane();

        Button myButton = new Button("Add");

        borderPane1.setTop(hbox1);
        borderPane1.setCenter(myButton);

        int width = 300;
        int height = 300;
        Scene scene = new Scene(borderPane1,width,height);
        primaryStage.setScene(scene);

        primaryStage.show(); //show the Stage
    }

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

}
```

1.2.4 Incorporating Java code with JavaFX

Now that we have generated the fxml code from Scene Builder, we need to make it do something. As explained at the beginning of this chapter, we could incorporate JavaFX with many different languages, but we will focus on Java.

How do we do that?

Let's got back to our first set of code from the beginning of this chapter:

```
import javafx.application.Application;
import javafx.stage.Stage;

public class Ch1_1 extends Application {
    public void start(Stage primaryStage) {
        primaryStage.show();
    }

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

In order to utilize the GUI that we created in Scene Builder we need to import the fxml code into Java. There are very few changes we need to make.

The first is that we need to tell Java what the name of the fxml file is. In this case it is "Tutorial1.fxml." Here is the code:
`FXMLLoader loader = new FXMLLoader(getClass().getResource("Tutorial.fxml"));`

NOTE: The file "Tutorial.fxml" and the Java file that you create must be in the same directory in the Operating System.

Next, we need to tell Java what class is in charge of the GUI. In this case, it is the "this" one - the one we are creating:
`loader.setController(this);`

Now, that we have gotten the fxml file and we told Java what class in charge, we need to load the fxml file:
`Parent root = loader.load();`

Now all we have to do is tell the Stage what Scene we want:

```
Scene myScene = new Scene(root,400,400);
primaryStage.setScene(myScene);
```

The following is the finished code including the correct imports:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.fxml.*;
import javafx.scene.*;
import javafx.stage.Stage;

public class Ch1_3 extends Application {
    public void start(Stage primaryStage) {
        FXMLLoader loader = new
            FXMLLoader(getClass().getResource("Test.fxml"));
        loader.setController(this);
        Parent root = loader.load();

        Scene myScene = new Scene(root,200,200);
        primaryStage.setScene(myScene);
        primaryStage.show();
    }

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

If we compile the Java code then it will not compile. The line `Parent root = loader.load();` reads a file so a `IOException` might be thrown. The easiest way to deal with this at this point is to add `throws Exception` after the "start" method. The following code works:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.fxml.*;
import javafx.scene.*;
import javafx.stage.Stage;

public class Ch1_4 extends Application {
    public void start(Stage primaryStage) throws Exception {
        FXMLLoader loader = new
```

```
            FXMLLoader(getClass().getResource("Tutorial1.fxml"));
        loader.setController(this);
        Parent root = loader.load();

        Scene myScene = new Scene(root,400,400);
        primaryStage.setScene(myScene);
        primaryStage.show();
    }

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

If we then run it then we get the figure 1.14.

Figure 1.14: What our GUI looks like with Java running it with a width of 400 pixels and a height of 400 pixels.

What happens when you click on the button? Nothing.

Why? Because you never told it to do anything. What did you expect it do? It looks nice though, even though it does nothing.

How do we connect the fxml controls to Java?

First, let's connect the Button in the GUI to our Java code. Right below the class definition, we are going to add `Button myButton;`. Why call it `myButton`? Because that is the id that we gave it back in Scene Builder. Look at figure 1.12 as a reminder. In what is circled in red is "myButton." If you named it something else in Scene Builder then you need to name it the same thing in Java.

Adding `Button myButton;` isn't quite enough though. Besides adding the import of `import javafx.scene.control.*;` we need to tell Java that this particular Button is coming from our fxml file, so we add `@FXML` before it, so that it looks like the following:
`@FXML Button myButton;`

When the line, `loader.load();` is run, it will try to connect `myButton` in Java and the .fxml file together.

Now that we have access to the button, let's do something when the person presses the button. Remember when we added a method to the `myButton` in Scene Builder and called it `handleButton` (see figure 1.13)? Now we can utilize that. We can now add the following code:

```
@FXML protected void handleButton(ActionEvent event) {
System.out.println("do something!");
}
```

We will also have to add another import:

```
import javafx.event.*;
```

Our modified code now looks like the following:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.fxml.*;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.event.*;
import javafx.scene.control.*;

public class Ch1_5 extends Application {
```

```
    @FXML Button myButton;

    @FXML protected void handleButton(ActionEvent event) {
        System.out.println("do something!");
    }

    public void start(Stage primaryStage) throws Exception {
        FXMLLoader loader = new
            FXMLLoader(getClass().getResource("Tutorial1.fxml"));
        loader.setController(this);
        Parent root = loader.load();

        Scene myScene = new Scene(root,400,400);
        primaryStage.setScene(myScene);
        primaryStage.show();
    }

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

Go ahead and run it. It will look the same at figure 1.14, but when you press the button "do something!" is printed in the console.

In order to complete our program we need to also include the TextField from the fxml file into Java as well.

So, we add, `@FXML TextField myTextField;`

(To see how to manipulate TextFields yourself, you can see the official documentation at the following link: `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TextField.html`.)

Now that myTextField is part of Java, let's access it. When the user clicks the button, we will get the input, which will be a String:
`String input = myTextField.getText();`

Now we need to convert the String to an int:
`int intInput = Integer.parseInt(input);`

Finally, let's change the println statement, so that it prints out 5 times the input:
`System.out.println(intInput * 5);`

The following code shows the completed application:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.fxml.*;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.event.*;
import javafx.scene.control.*;

public class Ch1_6 extends Application {

    @FXML TextField myTextField;
    @FXML Button myButton;

    @FXML protected void handleButton(ActionEvent event) {
        String input = myTextField.getText();
        int intInput = Integer.parseInt(input);
        System.out.println(intInput * 5);
    }

    public void start(Stage primaryStage) throws Exception {
        FXMLLoader loader = new
            FXMLLoader(getClass().getResource("Tutorial1.fxml"));
        loader.setController(this);
        Parent root = loader.load();

        Scene myScene = new Scene(root,400,400);
        primaryStage.setScene(myScene);
        primaryStage.show();
    }

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

Go ahead and run it. Put an int into the TextField. The application will print out the input times five.

NOTE: If you put anything besides an int into the TextField then you will have exceptions.

Now that you have gotten then far, change the application. Bring up Scene Builder and add another TextField.

When the Button is pressed instead of printing the result to the console, have it display in the second TextField. You will want to use the `setText()` method.

Chapter 2

Common JavaFX controls and their most common methods

There are many JavaFX controls, so to list them all would be beyond the scope of this book. If you need to see the documentation for all the JavaFX controls, see the following link: `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/package-frame.html`.

In addition, here is tutorial on how to use many of them from Oracle: `http://docs.oracle.com/javafx/2/ui_controls/jfxpub-ui_controls.htm`.

The following are the most common methods that you will use as a beginning programmer for the following controls:

TextField and Label:

- getText(): `public final String getText()`
- setText(): `public final void setText(String value)`

CheckBox and RadioButton:

- isSelected(): `public final boolean isSelected()`
- setSelected(): `public final void setSelected(boolean value)`

Slider:

- getValue(): `public final double getValue()`
- setValue(double): `public final void setValue(double value)`

Chapter 3

Common JavaFX Errors and Their Fixes

`java.lang.IllegalStateException: Location is not set.`:

- You do not have the correct fxml file or the fxml file is not in the correct location. Both the .java file and the .fxml must be in the same directory.
- In your Java file you misspelled the fxml file.

`java.lang.RuntimeException: java.lang.reflect.InvocationTargetException`:

- You forgot to give a control in Scene Builder an fx:id.
- You did not line up your fx:id's. For example, in Scene Builder you either forgot to give a control an fx:id or you did not match the fx:id in Scene Builder and the one in Java exactly. So, if in Scene Builder you have "myButton" and in Java you had "@FXML Button myButton1;" then you have a problem.
- In java you did not include the "@FXML" in your code. For example, you have `Button myButton;`, but you need `@FXML Button myButton;`

`java.lang.reflect.InvocationTargetException Caused by: javafx.fxml.LoadException: Error resolving onAction='#doSomething', either the event handler is not in the Namespace or there is an error in the script.`

- There is an "onAction" method in the fxml file that is not in the Java file. NOTE: It has to be named the same in both files. In this instance it is called "doSomething()."

`java.lang.NullPointerException`

- One of your controls is null. Why? Probably because your fxml file is not up-to-date. Did you save your latest updates in Scene Builder?

Chapter 4

Graphics Introduction - How to Draw with Javafx

What exactly are "graphics?" According to the dictionary, a "graphic" is either of the following:

- shown or described in a very clear way
- relating to the artistic use of pictures, shapes, and words especially in books and magazines

In terms of "computer graphics" we are referring to the second definition. So, "computer graphics" are just artistic renderings on a computer. Since this isn't an art book, we will focus on how to create shapes with which the user can interact.

Once again, there are two approaches to using graphics in JavaFX: use pre-created shapes or draw your own.

However, regardless of which approach you take, you need to understand how JavaFX (and most Graphics toolkits) use the origin.

4.1 Understanding the Origin

What is the "origin?" In geometry, the "origin" is where (0,0) is. It is the point where X=0 and Y=0. For example, look at figure 4.1.

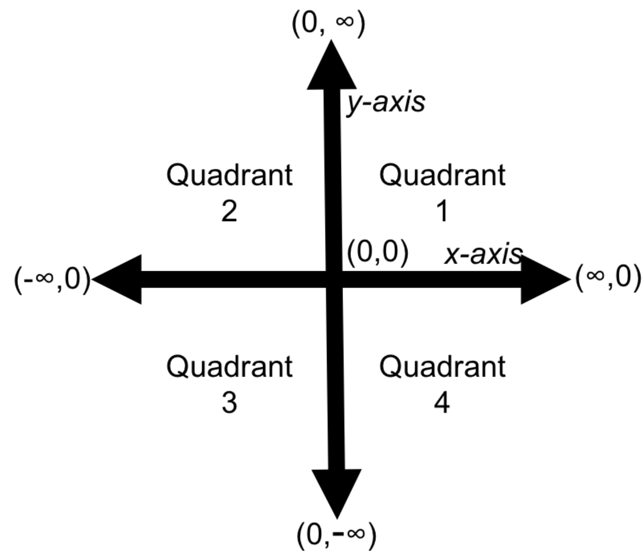

Figure 4.1: The traditional Cartesian coordinates with the origin in the center.

Figure 4.1 shows the traditional view of the Cartesian coordinates that is taught in the traditional geometry course. With the traditional Cartesian plane you have four quadrants. You have quadrant 1 (X and Y are both positive), quadrant 2 (X is negative and Y is positive), quadrant 3 (X and Y are both negative) and quadrant 4 (X is positive and Y is negative).

Although this is the traditional way to depict coordinates it is cumbersome for computer graphics. What is used instead is shown in figure 4.2.

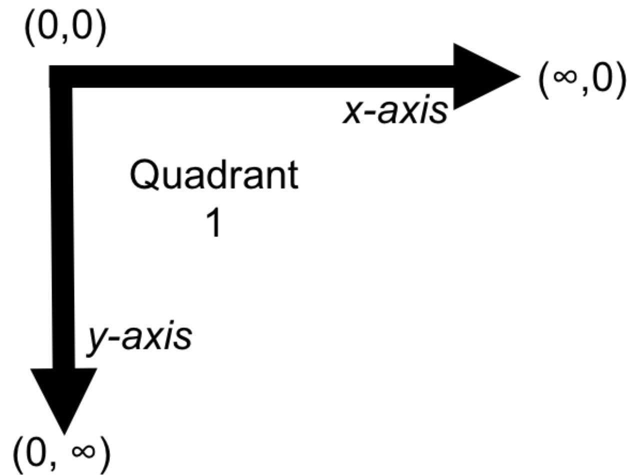

Figure 4.2: The coordinates used in computer graphics with the origin in the upper-left corner.

Figure 4.2 shows a simplified coordinate system. It only has one quadrant and the origin (0,0) is found in the upper-left corner.

This difference can be confusing at first since as *Y-axis* increases it goes down. However, there are several redeeming attributes to the modified coordinate system.

First, since there is only one quadrant you never have to worry about crossing an axis.

Second, there are no negative numbers - sort of. You can still use negative numbers, but their position will be off the screen.

4.2 Approach 1 - Use Pre-Created Shapes

4.3 Approach 2 - Draw Your Own Shapes

When you draw your own shapes you have complete control over what you draw.

That is good because if a shape does not exist then you can just draw it with a set of pre-defined methods in the API. As long as you understand what you want to draw and

understand the geometry behind it, then there is nothing you cannot do.

That is bad because you have to think of every detail of the shapes that you want to draw.

Chapter 5

How to make an executable Java program

One of the common desires among students is the ability to distribute their application to their relatives and peers. In order for your Java program to run on another computer there are two things that you need to do:

- Make sure that the target machine (the one that will run your Java application) has the Java JRE (Java Run-time Environment). The url for the JRE is the following: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. If the target machine already has the Java JDK (Java Development Kit) then it is fine because the JDK includes the JRE.
- Tell the person how to run your program. The easiest way is to provide an executable JAR file to the user. An executable JAR file is simply a Java file that you can double-click to run. It provides greater ease to the user.

5.0.1 Executable JAR file in JGrasp

Perform the following steps to create an executable JAR file in JGrasp:

1. Open JGrasp.
2. Create a Project for the application: From the menu: Project → New. Type in a project name. See figure 5.1.

3. After you click “Next,” checkmark the “Add Files to Project Now” checkbox, see figure 5.2 and figure 5.3.

4. Make sure that the project has the files that you need, see figure 5.4.

5. To create the executable JAR, from the menu select Project → Create JAR or Zip File for Project.

When complete, you will have a JAR file named after the project. For example, I named my project “ExecutableJARExample” so now I have “ExecutableJARExample.jar.”

NOTE: When you are using Scene Builder and fxml files then you have to add the fxml files to the project so that they are included in your executable jar.

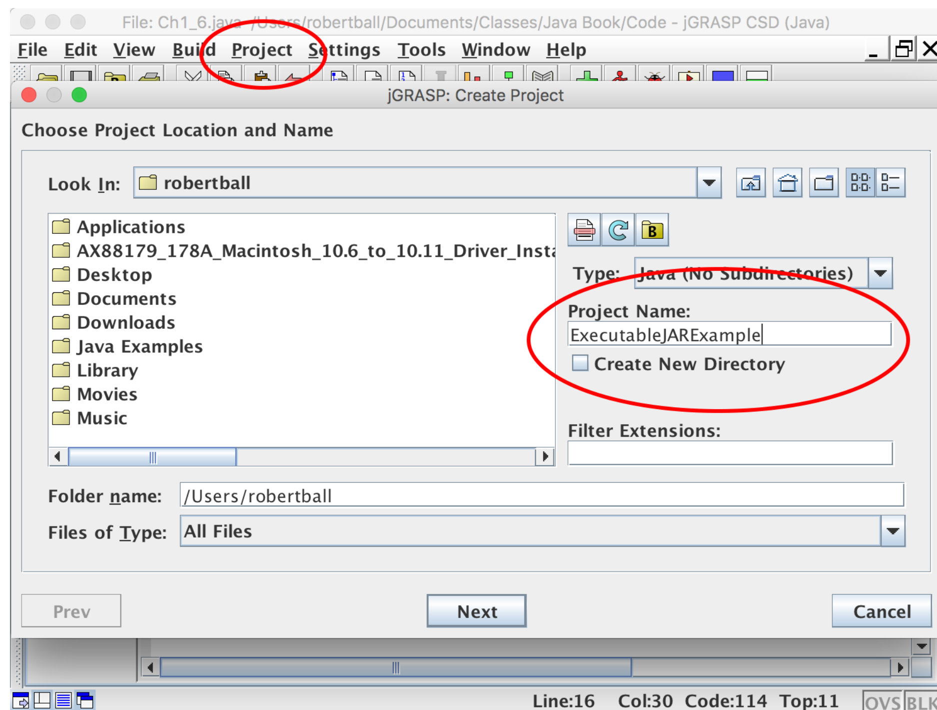

Figure 5.1: Creating a new Project in JGrasp: From the menu: Project → New. Type in a project name.

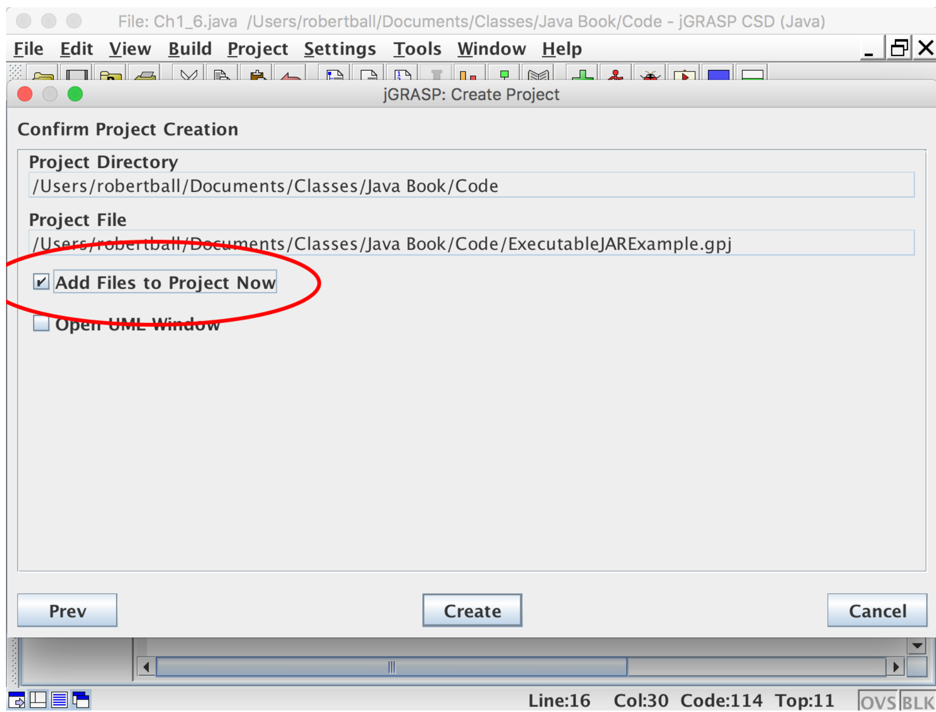

Figure 5.2: Click "Add Files to Project Now" - circled in red.

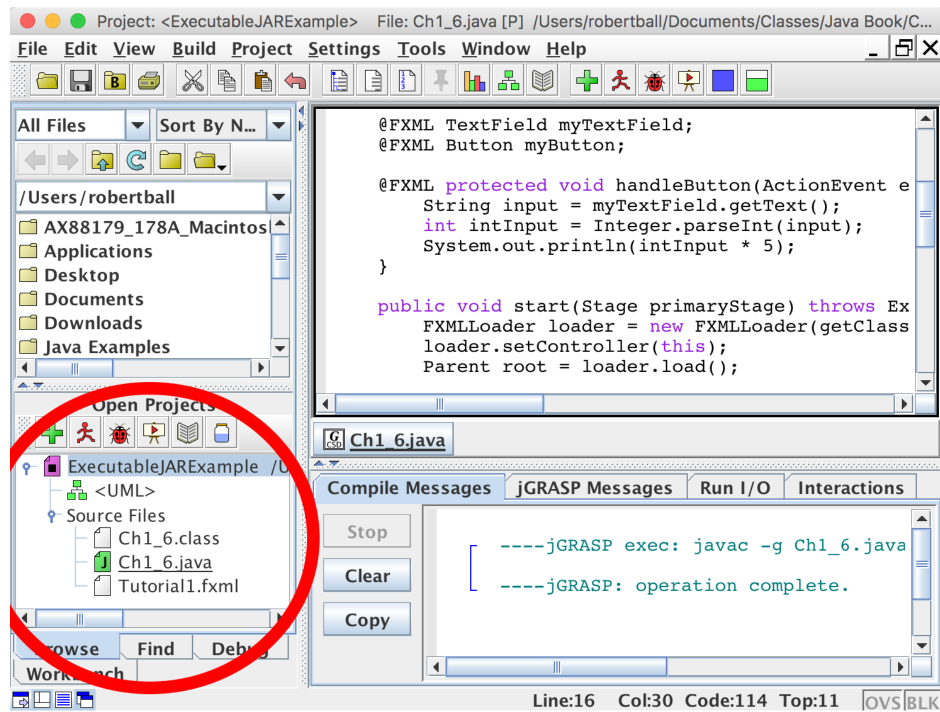

Figure 5.3: Once the Project is created, you can see the files that have been added in the lower left corner - circled in red. You can always add more files later by right clicking on the project name (currently highlighted) and selecting “Add Files.”

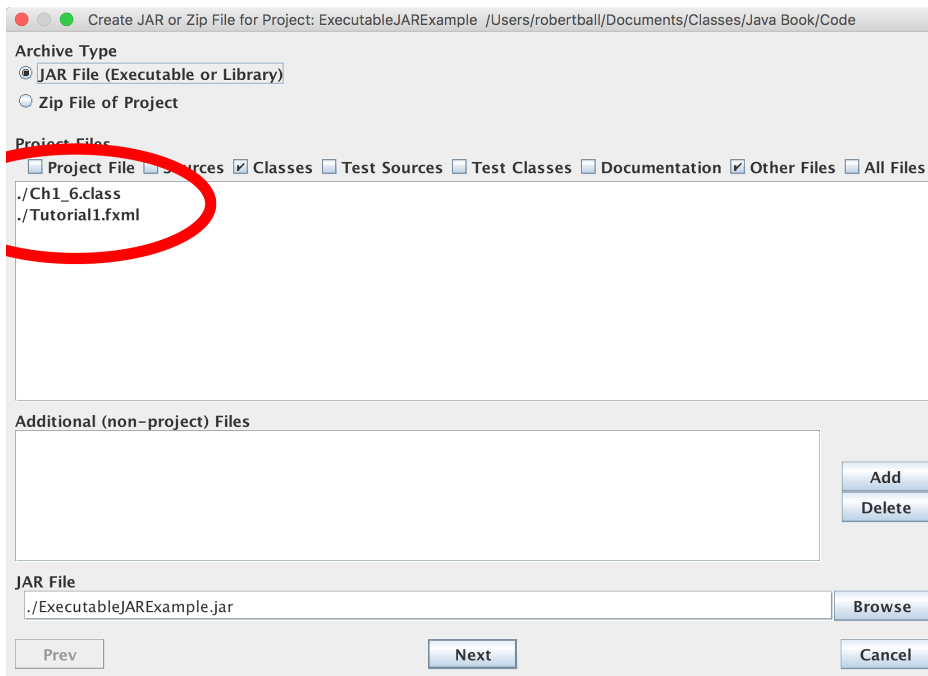

Figure 5.4: To create the executable JAR, from the menu select Project → Create JAR or Zip File for Project.

Index

BorderPane, 6

CheckBox, 25
computer graphics, 29
control, 6, 25

executable JAR file, 33

graphics, 29
GridPane, 6

HBox, 6

JavaFX, 1
JDK (Java Development Kit), 33
JRE (Java Run-time Environment), 33

Label, 25

origin, 29

RadioButton, 25

StackPane, 2, 6
Swing, 1

TextField, 25

VBox, 6

widget, 6