For this tutorial, we are going to help you get started with JavaFX. Note that you need to be on a platform capable of displaying a graphical window for this to work well. Most should be, but if you are on an OIT VM (or other remote machine) the default way of displaying a graphical window is an exported X display. If you have this setup, it will work, but is typically slow unless you have a very fast network connection. You might consider a local development setup, or use of VNC (or something similar).

Our goal is to make an RPN (also called postfix) calculator. If you are not familiar with RPN, the idea is that you have a stack of numbers. Each time you enter a number you push onto the stack. Each operation pops from the stack, performs the operation, and pushes back onto the stack.

For example, an input of

```
4 3 + 2 *
```

would push 4 on the stack, then 3 on the stack. The + operator would pop 3 and 4 (as + needs two operands), compute 7, and push 7 on the stack. Then we would push 2 on the stack, and finally we would pop 2 and 7, multiply them, and push 14 on the stack.

We are doing RPN rather than infix ("normal") for simplicity: we do not need to worry about order of operations, and we have simpler state to track.

1 JavaFX setup in Gradle

First, let us create a project for our calculator. Go ahead and create a new directory, initialize it as a git repository, and run gradle init (same settings as usual). Once you have the gradle project setup, follow the directions here:

https://openjfx.io/openjfx-docs/#gradle.

Note: please go ahead and include 'javafx.fxml' in your modules, since we will be using it later. Your modules line should read:

```
 modules = [ 'javafx.controls', 'javafx.fxml' ]
```

While you are at it, go ahead and add Mockito and TestFX to your dependencies section (we will make use of them later):

```
testCompile "org.mockito:mockito-core:2.+"
testCompile "org.testfx:testfx-core:4.0.16-alpha"
testCompile "org.testfx:testfx-junit5:4.0.16-alpha"
testCompile group: 'org.hamcrest', name: 'hamcrest', version: '2.1'
```

After the edits to the build.gradle file, this page references the source code from https://github.com/openjfx/samples/blob/master/HelloFX/Gradle/hellofx/src/main/java/HelloFX.java. You can just put that into the App.java created by gradle (except change the class name from HelloFX to App), like this:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class App extends Application {

    @Override
    public void start(Stage stage) {
        String javaVersion = System.getProperty("java.version");
        String javafxVersion = System.getProperty("javafx.version");
        Label l = new Label("Hello, JavaFX " + javafxVersion +
                            ", running on Java " +
                            javaVersion + ".");
        Scene scene = new Scene(new StackPane(l), 640, 480);
        stage.setScene(scene);
        stage.show();
    }

}
```

You should now be able to run in gradle and get a window that says the JavaFX version.

I'm going to take a few minutes to break this down a bit before we proceed. Our class extends Application which is the main class of a JavaFX application. Unlike "regular" programs, there is no main method. Instead, the entry point is start, which takes in the primary Stage. In JavaFX, a window is called a Stage (think of "stage as in where a play is" not "stage as in a step").

The code gets a few properties which are the JavaFX and Java versions (since that is what it will display). It then makes a Label. A Label is a UI component for displaying text (it does not do anything, just is a string shown on the screen). The constructor argument is the string to display.

In the next line, two things happen:

```
        Scene scene = new Scene(new StackPane(l), 640, 480);
```

The first is creation of a new StackPane. A StackPane is a layout. In general, a layout says how to display components together. In the particular case of a StackPane, it displays things back to front (on top of each other). Since we only have one thing to display (the Label l) the particular layout is not doing much.

We then pass this StackPane as the root of the Scene we are creating. The Scene is created to be 640 pixels wide by 480 pixels tall[1]

[1] I remember a time when this was considered a large resolution for an entire screen!

Finally, the Scene we just created is set as the scene of the Stage, and the Stage is shown.

If you need links to the documentation for the components in this section, they are

Application https://openjfx.io/javadoc/15/javafx.graphics/javafx/application/Application.html

Label https://openjfx.io/javadoc/15/javafx.controls/javafx/scene/control/Label.html

Scene https://openjfx.io/javadoc/15/javafx.graphics/javafx/scene/Scene.html

Stage https://openjfx.io/javadoc/15/javafx.graphics/javafx/stage/Stage.html

2 Layout and Component Basics

I'll note that while we typically want to start with the minimal piece of state, our goal in this tutorial is on the UI side, so we are going to start with some UI elements. The state of our program is actually quite simple: it is a string (for the current number we are entering) and a string of doubles (for our operand stack).

2.1 Adding Buttons

Our first goal is to setup the buttons for our calculator. Our goal is to make them look like this:

```
+---+---+---+---+
| + | - | * | / |
+---+---+---+---+
| 7 | 8 | 9 | E |
+---+---+---+ n |
| 4 | 5 | 6 | t |
+---+---+---+ e |
| 1 | 2 | 3 | r |
+---+---+---+---+
|   0   | . |   |
+---+---+---+---+
```

Now we need to think about a layout. Note that later we would want to put this grid of buttons as a smaller part of a larger design, but for now, we will just make this all we draw in our window.

A good choice for a layout here is a GridPane (https://openjfx.io/javadoc/15/javafx.graphics/javafx/scene/layout/GridPane.html). As its name suggest, a GridPane lays out its children in a grid. We can specify the row/column of a child when we add (if we do not specify, the default is the next available spot). We can also specify how many rows/columns a component should span. In our example above, most buttons span 1 row and 1 column (which is the default). However, our "Enter" button spans 3 rows, and our "0" button spans 2 columns.

We could do something like

```
GridPane gp = new GridPane();
gp.add(new Button("+", 0,0));
gp.add(new Button("-", 1,0));
gp.add(new Button("/", 2,0));
...
gp.add(new Button("1", 0,3));
gp.add(new Button("2", 1,3));
...
```

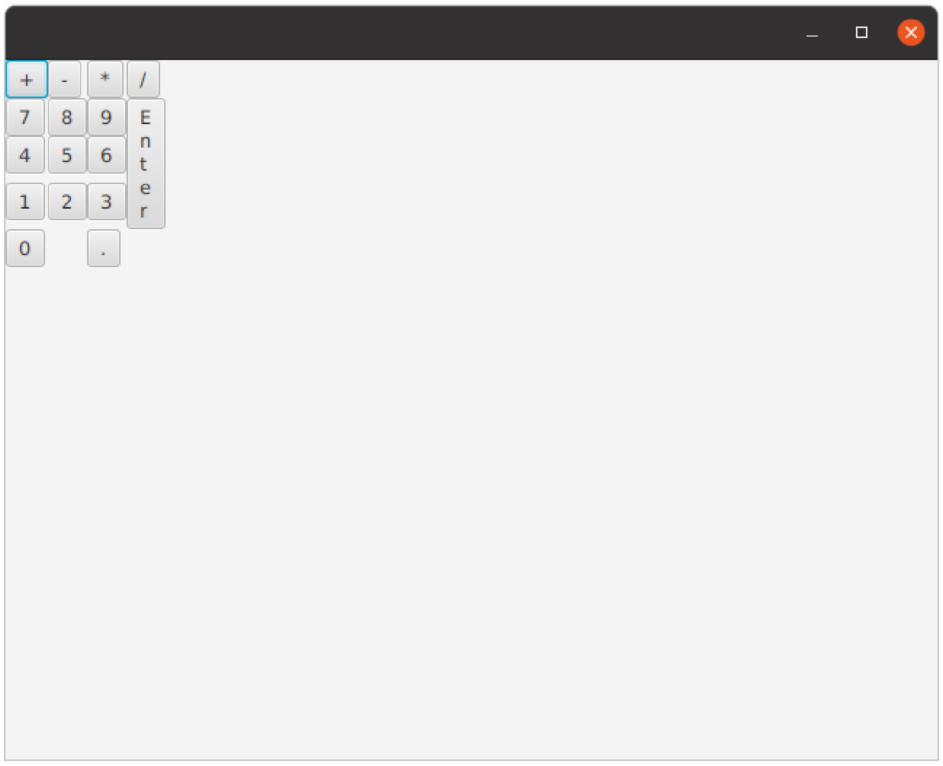
but that involves a lot of repetition of code. Instead, we might do this:

```
GridPane gp = new GridPane();
String[] labels= new String[] {"+", "-", "*", "/",
                               "7", "8", "9",
                               "4", "5", "6",
                               "1", "2","3",
                               "."};
int rows[] = new int [] {0, 0, 0,0,
                         1,1,1,
                         2,2,2,
                         3,3,3,
                         4};
int cols [] = new int[] {0,1,2,3,
                         0,1,2,
                         0,1,2,
                         0,1,2,
                         2 };
for (int i =0; i < labels.length; i++) {
  Button b = new Button(labels[i]);
  gp.add(b, cols[i], rows[i]);
}
gp.add(new Button("0"), 0,4,2,1);
gp.add(new Button("E\nn\nt\ne\nr"), 3,1,1,3);
```

Now, when we create our Scene, we can pass gp to its constructor as the root of the Scene:

```
Scene scene = new Scene(gp, 640, 480);
```

Let us take a moment to run this in gradle and see how it looks.

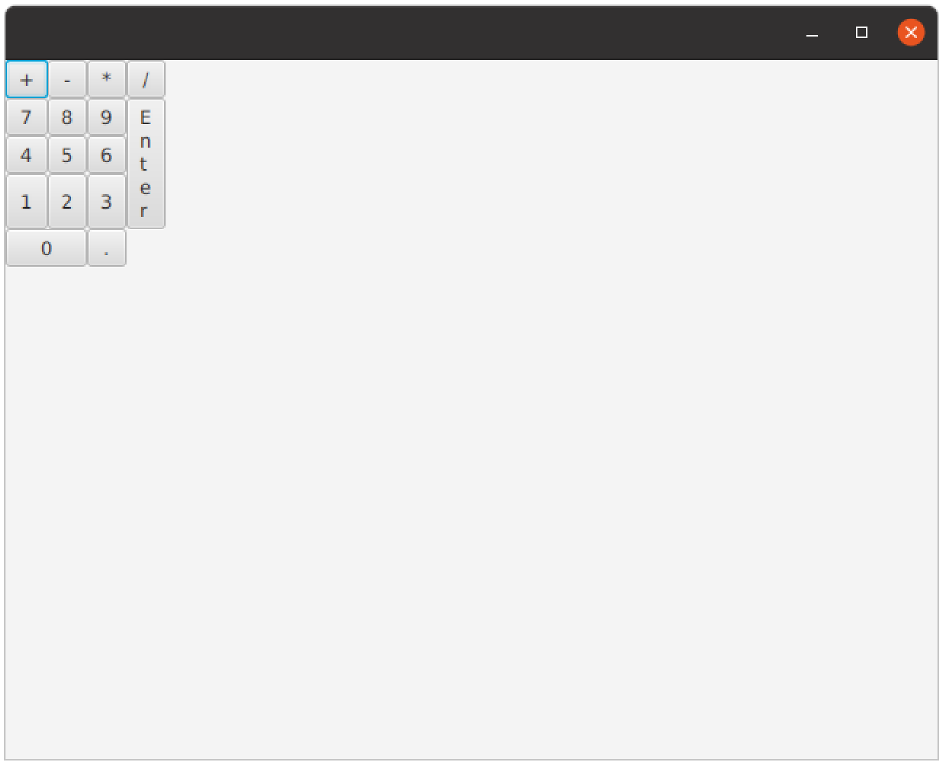


That looks pretty terrible... What is going on? The GridPane has laid our components out in a grid, but each of them is sized based on their own needs individually. For Buttons, the default sizing is basically just big enough to contain the label (and any image it has if you added one). Buttons by default do not grow bigger than their default sizing.

We could change this behavior by telling our buttons that they can get much bigger:

```
for (int i =0; i < labels.length; i++) {
  Button b = new Button(labels[i]);
  gp.add(b, cols[i], rows[i]);
  b.setMaxWidth(Double.MAX_VALUE);                    //Note this change
  b.setMaxHeight(Double.MAX_VALUE);                   //Note this change
}
Button b = new Button("0");
b.setMaxWidth(Double.MAX_VALUE);                      //Note this change
b.setMaxHeight(Double.MAX_VALUE);                     //Note this change
gp.add(b, 0,4,2,1);
b = new Button("E\nn\nt\ne\nr");
b.setMaxWidth(Double.MAX_VALUE);                      //Note this change
b.setMaxHeight(Double.MAX_VALUE);                     //Note this change
gp.add(b, 3,1,1,3);
```

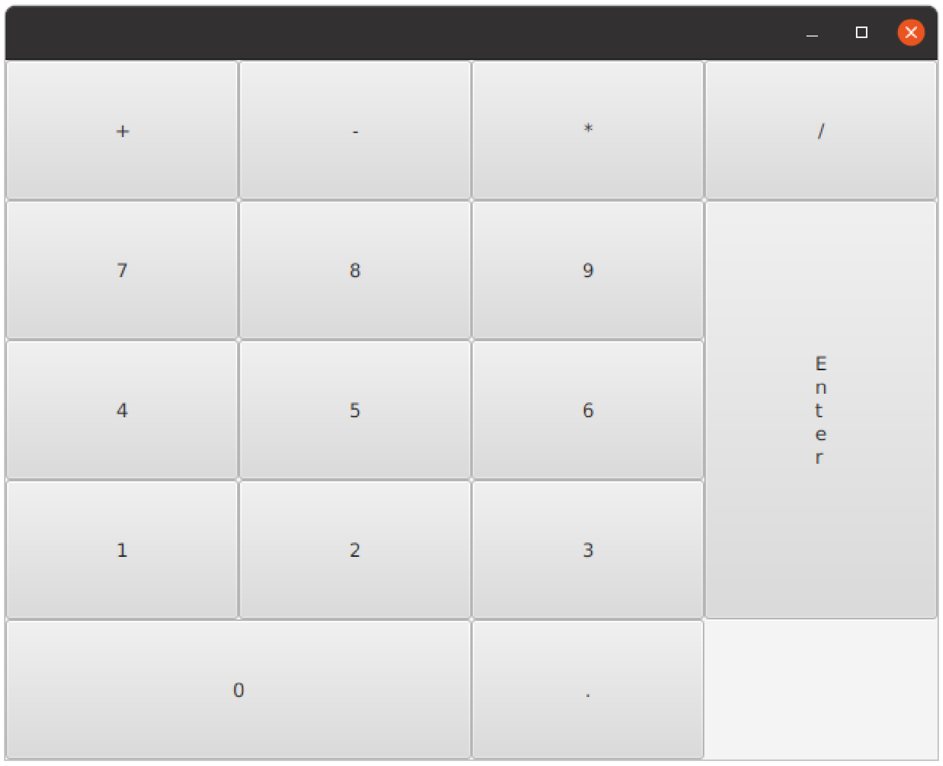
Here, we are creating the same buttons, but setting their max width/height to basically be unlimited (they can grow as much as the layout wants). Here is what we see now:



That actually looks much better, but still a bit strange. The 1,2,3 row is much taller than the others. Why is that? Basically the three number rows have to match the height of the Enter button, so absent any other instructions, GridPane made the last of them tall enough to meet that constraint. We can fix this problem by putting percentage contstraints on the GridPane's rows and columns:

```
for (int i =0; i < 4; i++) {
  ColumnConstraints cc = new ColumnConstraints();
  cc.setPercentWidth(25);
  gp.getColumnConstraints().add(cc);
}
for(int i =0; i < 5; i++) {
  RowConstraints rc = new RowConstraints();
  rc.setPercentHeight(20);
  gp.getRowConstraints().add(rc);
}
```

Now, if we run our code, we see:



This looks much better, in terms of an evenly laid out grid. Note that if you resize the window, it will resize the GridPane, and that will correspondingly resize all the buttons inside. If we were feeling really fancy, we might adjust the font size in each button based on the button's size. We aren't going to do that right now.

2.2 FXML

Instead of making the buttons fancier, let us notice that we have a huge mess of code in main. We could imagine refactoring the code: pulling this out into its own class, abstracting duplicate code into a method, etc. However, we are instead going to throw all that code away[2] and use FXML—a part of JavaFX that allows us to describe our component tree with XML (after all, XML is a great way to describe trees, and the GUI components are a tree).

From a code perspective, there is not much we do:

```
URL xmlResource = getClass().getResource("/ui/calcbuttons.xml");
GridPane gp = FXMLLoader.load(xmlResource);
```

This code asks the classloader to find the specified resource (relative to src/main/resources), and then the FXMLLoader to load it. The load method will return the root component in

[2]Why did we write all this code just to throw it away? It is instructive to see how things work when we write normal Java code before we switch to FXML

the XML file. Now we just have to create the calcbuttons.xml file. Copy our provided calcbuttons1.xml to src/main/resources/ui/calcbuttons.xml and open it up to take a look.

At the top you see some processing instructions (with the ?s):

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.RowConstraints?>
<?import java.lang.Double?>
```

The first of these is the XML prolog. The rest are import directives for FXML. You need them in the XML regardless of whether or not the names will be imported in the code that ultimately loads this file. Also, unlike “regular” java code, you need to import things from java.lang (*e.g.*,java.lang.Double).

Next, we see our top-level component:

```
<GridPane xmlns:fx="http://javafx.com/fxml">
```

The tag name starts with a capital letter, which tells FXML that you are creating an instance of a class. For the top-level item, we should include the xml namespace for the “fx” attributes.

Inside of that (before the closing `</GridPane> tag`) we have the various things that go inside of the GridPane. We have started out with our column and row constraints as we did above. Next, we have the children, which are all the Buttons that we want to make.

Looking at the first Button as an example, we see:

```
    <Button text="+" GridPane.rowIndex="0" GridPane.columnIndex="0">
      <maxHeight><Double fx:constant="MAX_VALUE"/></maxHeight>
      <maxWidth><Double fx:constant="MAX_VALUE"/></maxWidth>
    </Button>
```

Here, we are making a Button, setting its text to ”+”, and specifying its row/column index in the GridPane. Note that these are done with GridPane.rowIndex and GridPane.columnIndex instead of just rowIndex and columnIndex. This difference arises because the Button itself does not have a rowIndex or columnIndex. Instead, the GridPane needs to be told what row index and column index are used. We then specify that we want the maxHeight and maxWidth to be `Double.MAX_VALUE`. Here we need to use fx:constant to get a static constant inside of a class. The rest of the buttons are pretty similar, except that one has a rowSpan and the other has a columnSpan.

At this point, you should be able to run the code again and get the same behavior we had before: a nicely laid out grid of buttons which resizes when you adjust the window size.

You can read a lot more about FXML here `https://openjfx.io/javadoc/15/javafx.fxml/javafx/fxml/doc-files/introduction_to_fxml.html`

If you need more documentation for the FXML classes, you can find it at `https://openjfx.io/javadoc/15/javafx.fxml/javafx/fxml/package-summary.html`

2.3 CSS

I really do not like the fact we wrote out the max width/height for each button (not DRY!). We'd really like to abstract that out into a reusable style. JavaFX supports CSS (Cascading Style Sheets) to support this (and other) styling functionality. If you are not familiar with CSS, that is ok: we will give you a brief introduction to what you need to get started with JavaFX. Furthermore, JavaFX does not support the full set of features of CSS. If you want to see the documentation for JavaFX's CSS, it is here https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

Create the file `src/main/resources/ui/calcbuttons.css` and put this CSS in it:

```
.expandable {
    -fx-max-width:  infinity;
    -fx-max-height: infinity;
}
```

The first line says "we are going to define some styling for the class 'expandable'". Then everything inside the curly braces applies to that class. Inside the curly braces, we set `-fx-max-width` and `-fx-max-height` to infinity. Note that unless something is a standard CSS attribute, it will start with `-fx-`.

Now, let us change our XML file to use these classes instead of manually setting the max width/height. Copy calcbuttons2.xml to `src/main/resources/ui/calcbuttons.xml` and take a look at it.

You will see that we basically changed

```
    <Button text="+" GridPane.rowIndex="0" GridPane.columnIndex="0">
      <maxHeight><Double fx:constant="MAX_VALUE"/></maxHeight>
      <maxWidth><Double fx:constant="MAX_VALUE"/></maxWidth>
    </Button>
```

to this for each button (well really, I had Emacs automate the changes....)

```
<Button text="+" GridPane.rowIndex="0" GridPane.columnIndex="0" styleClass="expandable"/>
```

Now, we need to go back to our code and load the stylesheet. Right after we create our Scene, we need to do:

```
      URL cssResource = getClass().getResource("/ui/calcbuttons.css");
      scene.getStylesheets().add(cssResource.toString());
```

Now you should be able to run the program and it should once again look and act exactly like it did before.

Let us take a moment to make use of our CSS styling to make our buttons "fancier". In particular, let us make them a nice Duke Blue with white text, and a bit of a rounded edge. Let's add this style to our CSS:

```
.numbtn {
    -fx-background-color: black,#012169;
    -fx-text-fill: ivory;
    -fx-background-radius: 10, 10;
    -fx-background-insets: 2, 5;
}
```

Now, go into your XML file and change your number buttons (or whichever you want) to include this class (notice we added it in the styleClasses at the end, separated with a comma):

```
<Button text="7" GridPane.rowIndex="1" GridPane.columnIndex="0" styleClass="expandable,numbtn"/>
```

All of these background related properties are described in the "Region" section of the JavaFX styling documentation: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#region

Sadly, our current buttons do not show any difference when we click them (contrast with the buttons you have not styled). Not having a visual indication that you clicked the button is bad for usability. Let us fix that quickly, along with some visual feedback about which button the mouse is hovering over. Add this to your CSS:

```
.numbtn:hover {
    -fx-background-color: #333355, #101169;
    -fx-text-fill: ivory;
    -fx-background-radius: 10, 10;
    -fx-background-insets: 2, 5;
}

.numbtn:pressed {
    -fx-background-color: #000029, #000149;
    -fx-text-fill: ivory;
    -fx-background-radius: 10, 10;
    -fx-background-insets: 2, 5;
}
```

Now run your program again. The buttons should get slightly lighter when you mouse over them, and darker when you click them!

As a side note, if you want to see some examples of very nicely styled buttons, you can check out this page: http://fxexperience.com/2011/12/styling-fx-buttons-with-css/

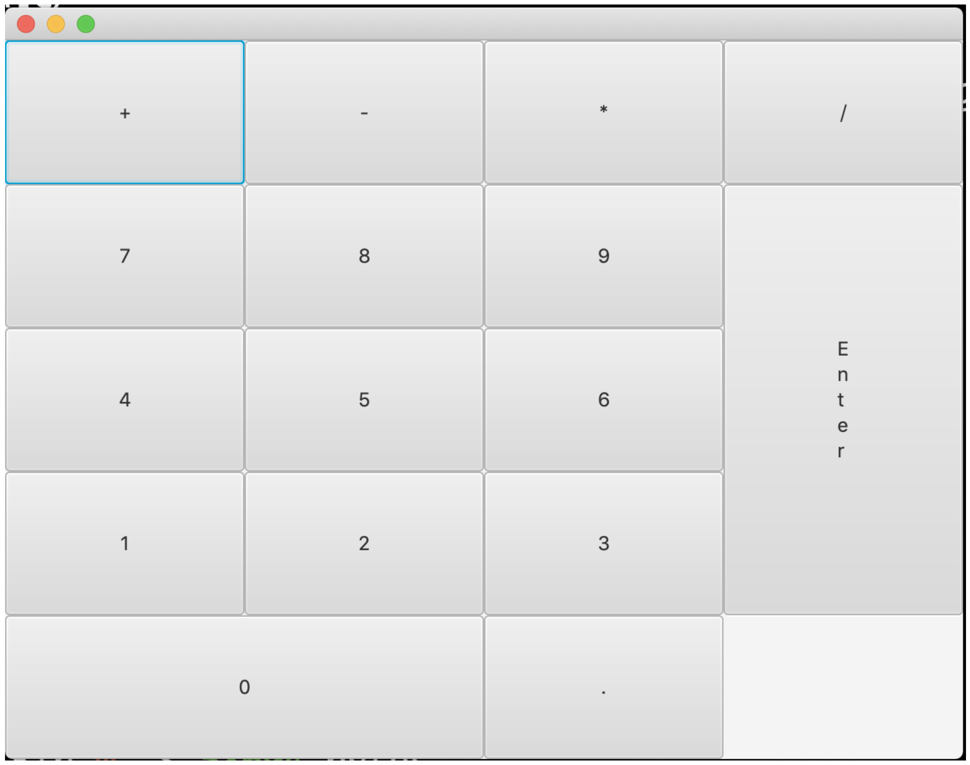
2.4 Testing Layout

As you have learned by now, testing is a critical part of development. How do we test this GUI code? We basically just did some manual testing and debugging on our button layout. We ran the code, saw that the buttons were not arranged well, adjusted the code, and repeated the process until we were happy with the layout. However, you should know by now that we want to automate our testing.

Testing things like a GUI layout is a bit tricky and different from testing functional behavior. For one thing, specifying the right answer is rather difficult. You might say "well the right answer is that I get exactly the image we saw above." On the one hand, this approach has some characteristics we want in testing. If we specify that the above layout is exactly what we want, and take an screen capture of it, then we automate our testing. We could run our code (in JUnit), programatically capture an image of what is displayed, and compare pixel by pixel. With such a setup, we could then use our test cases when *e.g.*, refactoring the code—any change that alters what is drawn is a regression bug and we should fix it.

However, there are a few problems with this approach. The first is a bit more philosophical: we can not realistically specify the right answer before we run our code. When we do functional testing, we can, before we even write factorial say that `fact(0)` should return 1, and `fact(4)` should return 24. Before running this code, could any of us draw a pixel-perfect image of what we expect the GUI to look like? The second problem is that there is no single right answer. There could be many subtle changes in how the GUI looks that we would be equally happy with (or perhaps happier).

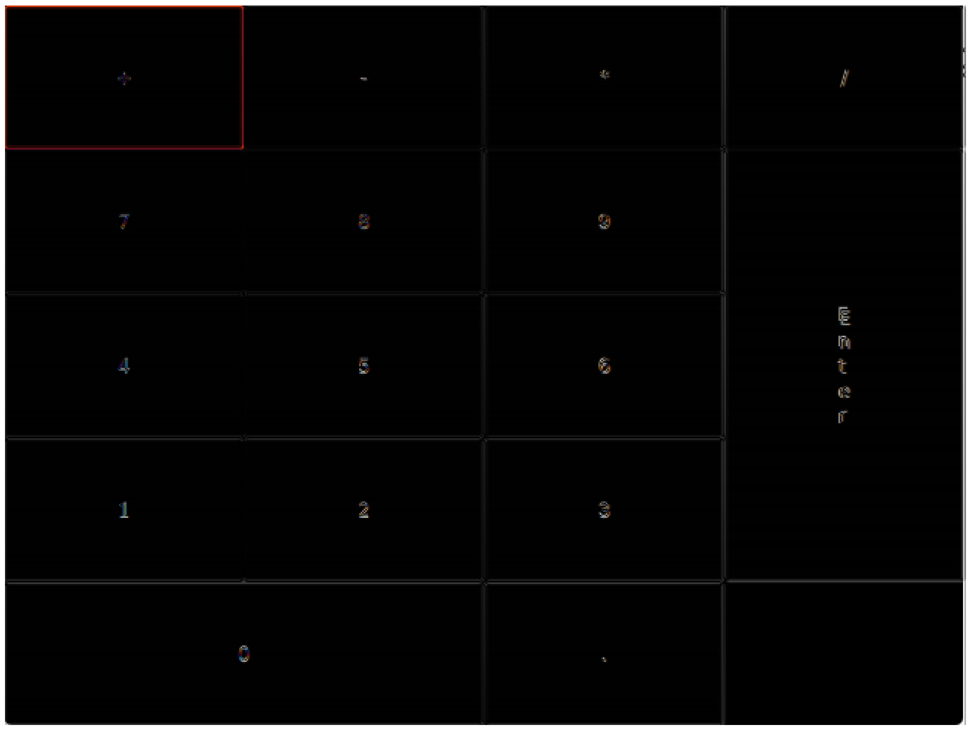
The third problem (which is closely related to the second problem) is that the specific image we get is not stable across platforms. The drawings above were captured on Ubuntu 20. However, if I run this on MacOSX (before applying the button color styling), I get:



You might think "hey those look the same" but there are some very subtle differences.

On the mac version, the top left button is highlighted with a blue box. The fonts and colors are also slightly different. The window bars on the top are also different, but we could easily ignore those and only look at the contents of the window.

If you are curious, you could see this a bit more clearly by examining the differences in the two images :



Here, black indicates the images were the same, and brighter colors indicate stronger differences. You can see that the buttons are mostly the same (they actually differ by very small amounts, so are close to black, but not exactly black). However, you can see the text mismatches, and differences in the lines between the buttons. We could imagine doing some image processing to figure out if the result is “close enough”. In particular, we could examine “differences” image above, and that we want to check that most pixels are “close to black” and that allow some variation in the middle of the buttons. This approach gets a bit murky in that we would have a hard time telling the difference between “slight differences in the font” and “we swapped the order of the buttons”. We might end up needing some crazy sophisticated image processing to test this well...☹

If we want to be very rigid in terms of what we accept, but are only concerned with the platform differences, we could have a correct image per platform. Our JUnit tests could then query what platform they are running on, and compare to the right set of images. A new platform would require a human to approve each image by hand. This approach could get cumbersome with changes (need a human to go through and recreate all images), and multiple platforms.

We note that in other contexts, an image-based approach may be much simpler and more desirable. If our GUI is supposed to draw a particular image (think Google Maps), then we can just check if that image (exactly) is drawn.

A third approach is that we could focus constraints: there should be a plus button in this region, and a minus button in that region. This sort of approach could give us a general idea that the GUI is laid out right, but becomes quite cumbersome to check all the finer details. Overall, checking the layout of a GUI in an automated way is quite difficult. Instead, we are going to focus on testing the functionality of the GUI automatically (and rely on a human to check the layout and aesthetics). To check the functionality, we first need our buttons to do something so that there is functionality to check. Let us do that first, then dive into testing it!

3 Controller for the number buttons

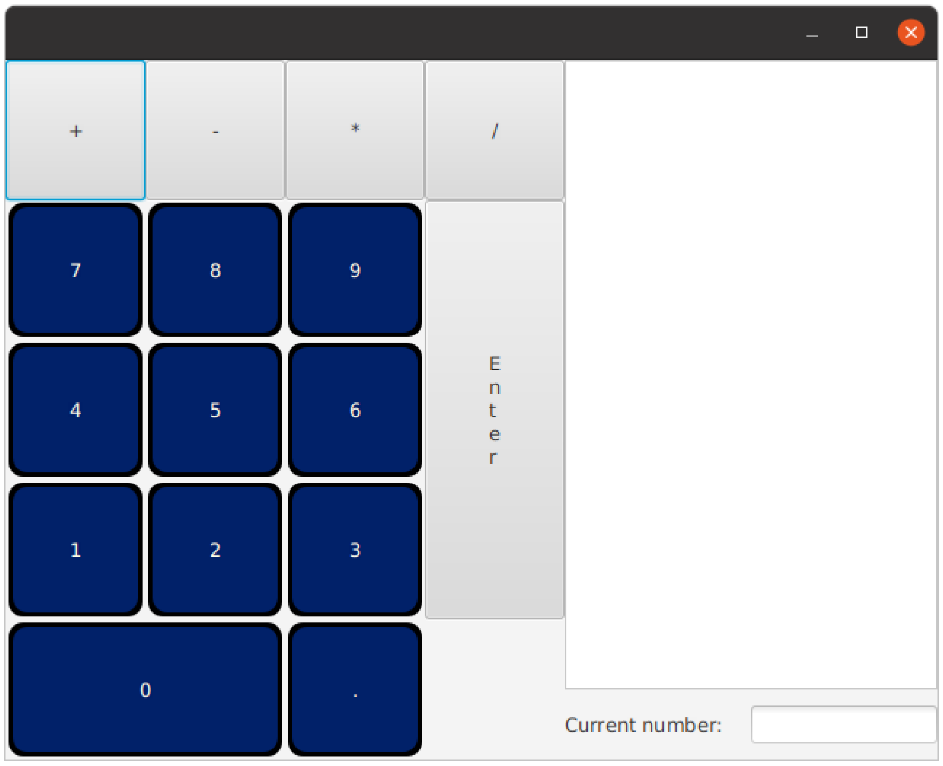
At the moment, our buttons do not do anything. To make them do something, we are going to need to make use of Controllers to handle our events. Before we do that, however, we need to have something for the buttons to do. To do that, we are going to need a bit more of our GUI. We might typically need a model (and will need one later). However, our number buttons are going to just edit a TextField, so there is no need for a complex model here.

3.1 The Rest of the GUI

Before we proceed, let us take a moment to put the buttons together with the rest of the calculator. Take a look at `calc-monolithic.xml`. This xml file puts a grid layout with 3 columns and 2 rows. Its first child is our GridPane from before with our buttons. That GridPane spans 2 rows. After that GridPane, we put in a ListView (where we will later display the operand stack), a Label, and a TextArea (to have the current number). The ListView spans 2 columns, where each of the components under it are a single column.

You may have noticed that we put `-monolithic` in the name of this XML file. That is because I have put the entire calculator into 1 XML file. Later we are going to break it into two pieces and use the `fx:include` directive. However, doing so is going to require a few extra concepts on the controller. Accordingly, to start out we are going to just put everything in one file, learn how to make a simple controller, then break it up as an improvement.

At this point, if you run your GUI, it should look like the following:



Of course, there may be some differences if you styled your buttons differently (or made other changes). Note that clicking on the buttons still does nothing—which we are about to fix next.

3.2 Your First Controller

Let us start by making a controller that only handles the number button presses (we'll include "." since we will treat that the same as a digit). When the user clicks a button, it will send an ActionEvent to the controller. That ActionEvent will specify which Button was pressed. Our controller's method for number buttons will take the text from the clicked button (*e.g.*, "7") and add it to the TextField where we have the current number. To do this task, we are going to write the following controller (in `src/main/java/edu/duke/ece651/calc/controller/NumButtonController.java`).

```
public class NumButtonController {
  @FXML
  TextField currentNumber;

  @FXML
  public void onNumberButton(ActionEvent ae) {
    Object source = ae.getSource();
    if (source instanceof Button) {
      Button btn = (Button) source;
      currentNumber.setText(currentNumber.getText()+btn.getText());
    }
    else {
      throw new IllegalArgumentException("Invalid source " +
                                          source +
                                          " for ActionEvent");
    }
  }

}
```

There are a few things going on in this class, some of which may seem odd. The first is that we have a TextField called currentNumber. You may recall that TextField is the JavaFX component we put in our XML file (in the lower right of our GUI) for the current number. It looks like currentNumber is uninitialized in this code, which may seem problematic. However, the FXMLLoader is going to initialize it for us. If you look in our XML, the fx:id of the TextArea is also currentNumber. As its fx:id matches this field's name, and their types match, the FXMLLoader will initialize the field to that component. We put an `@FXML` annotation on this field for two reasons. The first is that this annotation lets the loader modify the field even if it would otherwise not be able to due to access restrictions (like private). The second is that it gives the reader a hint that we expect this to be initialized by the FXMLLoader.

The second member of this class is the `onNumberButton` method. This method takes an ActionEvent as a parameter, which describes a button press. It first gets the source of that ActionEvent—the button that was pressed. It checks if that source is an instanceof Button. While we generally do not like to check instanceof and cast, we do not have a choice here. The source of an ActionEvent is an Object, and that does not have enough information. Inside the if, we cast to a Button. Now we can get the text of the Button and append it to the text of currentNumber.

What should we do if the source is not a Button? This situation *should* never come up: we are only going to hook this up to the event handler for Buttons. We'll throw an exception to indicate the problem in case we somehow write buggy code that allows this problem to happen. As a side note, we'd love it if the type system could prevent this sort of mistake, but adding this feature within the Java typesystem would be quite difficult.

Before we hook this up to our GUI (*i.e.*, make it so the buttons use it when we press them), let us take a moment to unit test it.

Here is our first step towards unit testing this controller:

```
@ExtendWith(ApplicationExtension.class)
public class NumButtonControllerTest {
  private TextField testText;
  private NumButtonController cont;
  @Start
  private void start(Stage stage) {
    testText = new TextField();
    cont = new NumButtonController();
    cont.currentNumber = testText;
  }
  @Test
  public void test_onNumberButton(FxRobot robot) {
    Platform.runLater(()->{
        Button b = new Button("7");
        cont.onNumberButton(new ActionEvent(b, null));
      });
    WaitForAsyncUtils.waitForFxEvents();
    FxAssert.verifyThat(testText, TextInputControlMatchers.hasText("7"));

  }
}
```

We are going to take a bit longer than usual to examine this test code, as there are several new things going on here. First, notice the `@ExtendWith`. This extends our class with the functionality of TestFX's ApplicationExtension class [3]. There are a few important things about doing this. First, if we just wrote "normal" tests, we would get an exception about the JavaFX toolkit not being initialized the first time we try to do anything. The second is that the `@Start` method will run on the JavaFX thread. JavaFX gets unhappy about many operations being done on other threads, as it is not thread safe. The second is that we will have access to an FxRobot in our test methods. We aren't going to use that in this case, but we will use it later. The FxRobot lets us automate UI interactions like "click this button" or "write this string in that text box."

Next, notice that we have some instance variables. We have not typically done that in our test code before, but do it now, as we need to communicate state from the `start` method to the test method(s). We have

```
@Start
private void start(Stage stage){
```

This method will get called before each test (arranged for by ApplicationExtension). Here we setup some components to use in our test. Note that in this example there is nothing wrong with moving any of the code here into the test method. However, more generally, anything that needs the Stage, or that is common to *all* test methods is best put here.

Let me take a moment to tell you just how important that `@Start` annotation is: If you do not put it on there, the method will not be run at the start of each test. This means any

[3] it is kind of like inheritance, not quite the same.

setup you did in it won't happen. I spent a while thinking I was going insane as my tests (in the next section) could not find the right components, and it was all because I found `@Start`.

Our `@Test` method takes an FxRobot, which we are not using here—we will learn more about that soon. Instead, we start with a call to `Platform.runLater`. This method takes a lambda which specifies what you want to do, and then runs that "sometime later" on the JavaFX thread. In this case, we want to do our manipulation of the various JavaFX components on that thread, so we make a button, use it as the source of our ActionEvent, and call our controller's method.

The line after `Platform.runLater` is a call to `WaitForAsyncUtils.waitForFxEvents`. This method waits until JavaFX has processed all the pending events in its queue—the one we care about being the one we put in the `Platform.runLater` call above[4].

After that, we use `FxAssert` to check that our `testText` component has the text `"7"` in it. FxAssert is part of TestFx and works around the idea of "Matchers". Here, we use TextInputControlMatchers (as we are working with a text-based input control) to see if it `hasText("7")`. See `http://testfx.github.io/TestFX/docs/javadoc/testfx-core/javadoc/org.testfx/org/testfx/matcher/control/package-summary.html` for the other matchers.

Note that we are calling the method directly here as we want to test the controller's functionality in as much isolation as possible. We want to decouple that from the construction of the GUI and connection of the controller to the GUI's components (which we will do shortly). You might be thinking "but is this not exactly what mocking is for?" It is exactly what mocking is for, and we'd *love* to mock if we could. Sadly, many of the methods we need in the JavaFX components are *final* so we cannot mock them.

Note that the one test case we did above is rather insufficient. What if, for example, we concatenated the strings backwards (old + new vs new + old)? Our single "button click" test case would not show this problem. We'll do a little refactoring to pull out the "make a button and click it" code (along with the things to run it on the JavaFX thread and wait) , and write a second test case:

```
@ExtendWith(ApplicationExtension.class)
public class NumButtonControllerTest {
  private TextField testText;
  private NumButtonController cont;

  @Start
  private void start(Stage stage) {
    testText = new TextField();
    cont = new NumButtonController();
    cont.currentNumber = testText;
  }
```

[4]I will note that for what we are doing here, we *could* get away with doing it on the testing thread rather than the JavaFX thread, but that is a bad habit to get into.

```
  private void addNums(String ... strs) {
    Platform.runLater(()->{
        for (String s: strs){
          Button b = new Button(s);
          cont.onNumberButton(new ActionEvent(b,null));
        }
      });
    WaitForAsyncUtils.waitForFxEvents();
  }

  @Test
  public void test_onNumberButton_7(FxRobot robot) {
    addNums("7");
    FxAssert.verifyThat(testText, TextInputControlMatchers.hasText("7"));
  }

  @Test
  public void test_onNumberButton_pi(FxRobot robot) {
    addNums("3", ".", "1", "4");
    FxAssert.verifyThat(testText, TextInputControlMatchers.hasText("3.14"));
  }

}
```

As a quick side note: If you try to run any of the TextFX tests in the debugger, they won't work. JUnit reports that it cannot discover any tests, and the debugger exits. I will look into this problem: for now, just do not write any bugs in your code ☺. More seriously, TestFX does come with a tool called DebugUtils, which lets you dump information (including screen shots) on test failure.

3.3 Hooking Up Our Controller

Now, we need to hook this controller up to our UI code. To do so, we need to change two things in the XML. First, we need to specify that this class is the Controller. To do so, we have to put an `fx:controller` attribute on the root element. Note that we can only specify `fx:controller` on the root element of an XML file. For this particular file, we change the first GridPane to:

```
<GridPane xmlns:fx="http://javafx.com/fxml"
          fx:controller="edu.duke.ece651.calc.controller.NumButtonController">
```

Note that we provide the fully qualified name of the controller class here.

The second change we need to make is to connect the event handlers of particular components to the methods in this controller. In our case, we are using Buttons and want to set their onAction event (which is what happens if you click the button or hit Enter if it is selected) to the onNumberButton method in the controller. Accordingly we add `onAction="#onNumberButton"` to each button that we want to have this behavior (namely 0–9 and decimal point).

For example, the 7 button is now described as:

```
<Button text="7"
        GridPane.rowIndex="1"
        GridPane.columnIndex="0"
        styleClass="expandable,numbtn"
        onAction="#onNumberButton"/>
```

You can see these changes all made in `calc-monolithic-controller.xml`. Put that in your resources directory, and change the name of the XML file you load in your App.java, then run the calculator. You should now be able to click the number buttons and have them add to the TextField in the lower right.

You have just tested this by hand, but we'd like to have an automated integration test (making sure the view and controller work together properly). This test is going to put together the whole GUI, use the FXRobot to test clicking on the buttons, and then check the text box for the right contents.

Let's head over to AppTest.java and write this test:

```
@ExtendWith(ApplicationExtension.class)
class AppTest {
  App a;
  @Start
  public void start(Stage stage) throws IOException {
    a = new App();
    a.start(stage);
  }

  @Test
  void test_numButtons(FxRobot robot) {
    FxAssert.verifyThat("#currentNumber", TextInputControlMatchers.hasText(""));
    String str = "123450.6789";
    for (char digit : str.toCharArray()){
      if (digit == '.') {
        robot.clickOn("#dot");
      }
      else {
        robot.clickOn(""+digit);
      }
    }
    FxAssert.verifyThat("#currentNumber", TextInputControlMatchers.hasText(str));

  }
}
```

Before we walk through the details of this test code, go ahead and run it (C-c C-t). You should see the GUI window pop up, and then see the buttons get clicked. As each gets clicked, the right number appears in the text box in the lower corner. Finally the window disappears. Note that while this is happening, you should not touch the mouse or try to change windows or anything like that. You will mess up the results of the testing if you interfere (and possibly do strange things to whatever other program you get in the way with).

Now let us dive into the details. First of all, we have our `start` method. This method creates our App, and calls its start method, passing in the Stage. We remember the App in an instance variable in case we need it for other purposes later.

Next, our test method does a few things. First it checks that the component whose fx:id is `currentNumber` has the empty string for its text. The `#currentNumber` is a CSS selector which finds a component. You can use most CSS selectors (see the FXML documentation for which ones you cannot use). The best one is generally the fx:id, since you should always make those unique. We can use the CSS selector instead of the component object itself with FxAssert.verifyThat.

Next, we have the string we want to put in. We iterate over the digits, and for each one, we ask the robot to clickOn the right button. For the digits, we use their text, for example ”7”. For the dot button we have to use an ID because "." is the start of a CSS class in a selector. Note that we could use an ID for the digits if we added an ID to each button in our XML. We are mostly just demonstrating different ways to select things here.

Note that the robot will take care of doing things on the right thread and waiting for them to finish. After we finish having our robot click, we check that our currentNumber TextField has the string we want.

You will use FxRobot for a lot of testing. The full documentation is here `https://testfx.github.io/TestFX/docs/javadoc/testfx-core/javadoc/org.testfx/org/testfx/api/FxRobot.html`

but some of its more commonly used behaviors are:

clickOn Go to the specified component and click on it.

moveTo Move the mouse to the specified location.

drag/drop Do drag and drop

write Type text into the current component

3.4 Refactoring our XML file

I previously mentioned that we would prefer to split our XML file up into two pieces, and use the `fx:include` directive. Not only would this make our XML files smaller and easier to work with, but it would also give us a re-usable component. With the buttons split out into their own panel, we could make use of them in a separate context. Another benefit, which we will see in this section is that we can use one controller per XML file. All elements in one XML file must share a controller. However, each different XML file uses its own controller.

We could start by cutting out the inner GridPane (and all of its children) and putting them in their own file. Doing so, we would need to put the appropriate `<?import>` directives into that file. We then need to replace it with a `fx:include` directive at the relevant place in the file we removed it from. This directive would look like:

```
    <fx:include source="calc-buttons-split.xml"
                GridPane.rowIndex="0"
                GridPane.columnIndex="0"
                GridPane.rowSpan="2"/>
```

Notice that we put the GridPane.xxx attributes onto this directive. Those will go onto the component that is included. We want them here, as those are particular to how the contents of calc-buttons-split.xml are placed into this layout. If we re-used calc-buttons-split in some other context, we might want those buttons in a different place in a GridPane, or they may not even be in a GridPane at all (they could be in some other layout).

We also need to move the `fx:controller` to the root of calc-buttons-split and put the xmlns on it:

```
<GridPane xmlns:fx="http://javafx.com/fxml"
          fx:controller="edu.duke.ece651.calc.controller.NumButtonController">
```

We have provided calc-split.xml and calc-buttons-split.xml. Take a moment to look at them and then switch your code in App.java to load calc-split.xml.

When you run it, your GUI should display, however, when you click a number button, you will get a NullPointerException. If you look at the stack trace, you will see that it starts with an InvocationTargetException, which means another exception happened during a reflective method call. Scroll down the stack trace, and you will see

```
Caused by: java.lang.NullPointerException
        at edu.duke.ece651.calc.controller.NumButtonController.onNumberButton(NumButtonC
        ... 57 more
```

You can see that the NullPointerException is in NumButtonController. What is going on? If you look at that line of code, you will see that it is make using of currentNumber. Remember that currentNumber is the TextArea field that we want to have initialized by the FXMLLoader. What is going on now?

The FXMLLoader fills in the fields after each file. When it does so, it has not finished the parent files, and will not use any of the components in them to resolve controller fields (especially as they may not have even been created yet). We can still make this work, we just have to do a little bit of connecting things up ourselves. In particular, we are going to need to make the CalculatorController (which will be the controller for our top-level XML file). It will have also have a field for the currentNumber, as well as one for the NumButtonController. It is also going to implement an interface called Initializable, and provide the `initialize` method required by that interface. After the FXMLLoader fills in the fields of a controller, it checks if the controller implements initializable. If so, it calls initialize, which allows the controller to do any initialization it needs to do after the fields have been filled in. Our CalculatorController looks like this:

```
public class CalculatorController implements Initializable {
  @FXML
  TextField currentNumber;

  @FXML
  NumButtonController numButtonController;

  public void initialize(URL location, ResourceBundle resources) {
    numButtonController.currentNumber= currentNumber;
  }

}
```

Could you guess that we are going to have the FXMLLoader fill in numButtonController for us (that is what the `@FXML` annotation should tell you!)?

The initialize method is very simple: all we are going to do is take our currentNumber (which will be filled in by the FXMLLoader) and puts it into numButtonController's currentNumber.

Now we just need to go edit the XML files to make this happen. First, in calc-split.xml, we need to specify this class as the controller:

```
<GridPane xmlns:fx="http://javafx.com/fxml"
          fx:controller="edu.duke.ece651.calc.controller.CalculatorController">
```

Then we need to add an `fx:id` to our include:

```
  fx:id="numButton"
```

Note that the id here is numButton but the field in the class is numButtonController. When you put an fx:id on an fx:include, it makes two name bindings: one for the component and one for the controller. Here, numButton would refer to the GridPane, and numButtonController would refer to the NumButtonController (maybe we should rename this as the former is not logical).

At this point, your program should work again. You should both be able to run it directly and use the number buttons, and to run the JUnit tests.

4 Finishing The Calculator

At the moment, our calculator "works" if you just want to type in a number, but not if want to do anything with that number (like add it to another). We have the View to support the calculator functionality (all the buttons and a place to put the operand stack). We need a model to support the calculator. This model is going to need to be a class that handles the RPN stack. It should have a stack of numbers, and support operations like "plus" and "times". We also need a controller to manipulate the model—the plus button should be hooked up so that it calls the add method. The controller requires a bit more than that: we actually want plus to take any current number (if there is one), and push it on the stack, then do plus.

4.1 Our Model

We will start with the model, which we are just providing for you. There is not much new or interesting to learn here. You can find it in the enclosed RPNStack.java file (which should go in src/main/java/edu/duke/ece651/calc/model). Before going any further, take a look at this file and observe that nothing in it is related to the GUI. If we were to want to use a text-based calculator, this model would be exactly the same. Some other code (the controller) will obtain the user's actions (numbers, pluses, minuses, etc) whether by button clicks or by typing in the terminal. That code will pass the requests to the model which will update its state. Then yet another piece of code (the view) will display the state.

We've also provided a few simple tests for this code in RPNStackTest.java (which should go in src/test/java/edu/duke/ece651/calc/model).

4.2 NumButtonController Revisited

Let us take a moment to head over to NumButtonController and make it work for plus. First we need to add a reference to the model. We're going to want to inject the dependency via the constructor. We'll add:

```
RPNStack model;

public NumButtonController(RPNStack model) {
  this.model = model;
}
```

Now, let us write the code to handle the enter button. Why start with enter? It is the simplest, we will want to re-use the code for it for our other buttons. In particular, we will write:

```
  void pushCurrentNumIfAny() {
    String s = currentNumber.getText().trim();
    if (!s.equals("")) {
      double d = Double.parseDouble(s);
      model.pushNum(d);
    }
    currentNumber.setText("");
  }

  public void onEnter(ActionEvent ae) {
    pushCurrentNumIfAny();
  }
```

The first method can be re-used in our other buttons to push the current number if there is one before doing the operation.

Before we do anything else, we would like to unit test our NumButtonController. Before we try to write any new tests, let us run our old ones... Your old NumButtonControllerTest does not compile now, as we have changed the constructor that it uses—it now needs an RPNStack. This is a great place to use mocking: we would like to isolate the model from the controller in testing. Accordingly, we will add a `private RPNStack model;` field to this class, and change our start to mock it and pass the mock to the NumButtonController constructor:

```
  @Start
  private void start(Stage stage) {
    testText = new TextField();
    model = mock(RPNStack.class);  //this is what is new here!
    cont = new NumButtonController(model);
    cont.currentNumber = testText;
  }
```

Now, if you try to run the tests, your code will compile, but your test in AppTest will fail with a really big exception. The short version is that we can no longer create this controller with the default constructor, which is what the FXMLLoader is trying to do. We are going to fix that soon, but want to unit test the controller before we make more changes. Go to AppTest.java and put `@Disabled` on the class declaration, like this:

```
@Disabled
@ExtendWith(ApplicationExtension.class)
class AppTest {
```

Putting `@Disabled` will make JUnit skip over this test class. When we have fixed the problems, we'll remove the `@Disabled` annotation to re-enable that class.

Now, let us return to NumButtonControllerTest and add in a test for our enter button:

```
@Test
void test_enterButton(FxRobot robot) {
  Platform.runLater(()->{
      testText.setText("1234.5");
      Button b = new Button("Enter");
      cont.onEnter(new ActionEvent(b,null));
    });
  WaitForAsyncUtils.waitForFxEvents();
  verify(mockedModel).pushNum(1234.5);
  verifyNoMoreInteractions(mockedModel);
  FxAssert.verifyThat(testText, TextInputControlMatchers.hasText(""));
}
```

In this test, we do a few things on the JavaFX thread: set the text field to 1234.5 and call the method in our controller will handle the enter button. On the test thread, we then verify that the controller called pushNum(1234.5) on the model. It also verifies that nothing else was done to the model, and that the controller cleared out the text field.

We'll let you write a test case on your own for if the text field contains only white space. In doing so, can you find some common code to abstract out?

4.3 Hooking Up This Controller Method

Now, lets go to our XML file for the buttons, and add the controller method to the enter button. In particular, we will add an `onAction` of `#onEnter`.

```
<Button text="${'E\nn\nt\ne\nr'}"
        fx:id="Enter"
        GridPane.rowIndex="1"
        GridPane.columnIndex="3"
        GridPane.rowSpan="3"
        styleClass="expandable"
        onAction="#onEnter"/>
```

It seems like we should be done, right? There is one remaining detail: our NumButtonController no longer has a default constructor! How will the FXMLLoader know how to create it? We are going to have to tell it somewhere.

Let's go back to our App.java and specify how to create this controller. First, we need to create our model.

```
RPNStack model = new RPNStack();
```

Then we are going to have to switch how we use FXMLLoader. Instead of the static load method, we will need to create an instance of FXMLLoader, passing in the URL to load:

```
FXMLLoader loader = new FXMLLoader(xmlResource);
```

Next, we will have to use loader's setControllerFactory to specify how to create controllers. Here, we need to pass in a function (lambda) that takes a Class and returns an Object. There are a lot of approaches to this, each with some tradeoffs. One straightforward way is to make a HashMap with the controllers we want and to have the controller factory look up the relevant class:

```
HashMap<Class<?>,Object> controllers = new HashMap<>();
controllers.put(NumButtonController.class, new NumButtonController(model));
controllers.put(CalculatorController.class, new CalculatorController());
loader.setControllerFactory((c) -> {
    return controllers.get(c);
});
```

This approach is simple and straightforward, but requires us to change the code for each new controller class we make. It also means that if, for some reason, we had multiple controllers of the same type, they would share an instance. That may or may not be our desired behavior.

We could imagine using reflection to create the controllers. Such an approach is more flexible in terms of adding other classes, but is complicated in other ways. For one thing, we need to figure out which constructor to use, and what information to pass to it. We could constrain all our controllers to have the same constructor signature (*e.g.*, all of them take one argument for the model), in which case we can just call that constructor. However, if we need to change what is passed, we must then change every constructor. We can imagine more and more complex approaches. For now, we will just use this simple one.

After that, you will use the load method on the loader to read the XML file:

```
GridPane gp = loader.load();
```

At this point, you should be able to run your calculator, and the enter button should clear out any text that is in the current number area, however nothing else shows up. Why is that? Our controller's method pushes the number onto the operand stack in the model, and clears the text. However, the view for the operand stack is not hooked up to the model.

Now that we have fixed how we load the XML, we can also return to AppTest and remove the `@Disabled` on the class. All our tests should pass now.

4.4 Displaying the Operand Stack

Our next step is to display the operand stack. We can do so by putting this code right after we load the Scene:

```
ListView<Double> operands = (ListView<Double>) scene.lookup("#rpnstack");
operands.setItems(model.getList());
```

You will see that the right side of that assignment is underlined in orange because we are making a cast without first checking that the resulting Node is actually a ListView. We generally do not like casts, but this is a case where we are pretty much stuck with it: we can only lookup components as Nodes, and we know that rpnstack should be a ListView. We could make this go away by writing an if that checks instanceof first. However, what would we do if it is not the type we expect? We would pretty much just need to throw an exception (which is what would happen without the check anyways). Writing this extra if would just clutter up the code. We also do not like to leave warning around either. Instead, we can add `@SuppressWarnings("unchecked")` right before that line to indicate that we know what we are doing[5]:

```
@SuppressWarnings("unchecked")
ListView<Double> operands = (ListView<Double>) scene.lookup("#rpnstack");
operands.setItems(model.getList());
```

Note that since we use an ObservableList, the list automatically notifies the view when it changes, and we do not need to do anything specially.

Now you should be able to run your calculator and use the “enter” button to push the current number onto the stack, which should display in the area on the right.

While we are at it, let us take a moment to go update AppTest (which is our integration test of these things) to make use of the Enter button and check that the list view is updated.

At the end of our current test (which clicks out the number 123450.6789), let us add

```
robot.clickOn("#Enter");
FxAssert.verifyThat("#currentNumber", TextInputControlMatchers.hasText(""));
FxAssert.verifyThat("#rpnstack", ListViewMatchers.hasItems(1));
FxAssert.verifyThat("#rpnstack", ListViewMatchers.hasListCell(123450.6789));
```

That is, we’ll have our robot click the enter button. Then we will check that it emptied out the currentNumber field. Next, we’ll check that the rpnstack ListView has 1 item, which is 123450.6789. I will note that hasListCell just checks for any cell, but not for ordering. That is somewhat unfortunate, as in later/more complex test we might like to check particular cells. In such cases, we’ll just have to get those things from the lists items ourselves.

4.5 Operation Buttons

Now we need to make the plus, minus, times, and divide buttons work. Let us start with plus. By now, you should be pretty familiar with the ideas. First, we add a method to our NumButtonController:

```
public void onPlus(ActionEvent ae) {
  pushCurrentNumIfAny();
  model.add();
}
```

[5]You might find that it seems like sometimes you can put a SupressWarnings before a single line of code and sometimes you cannot. The rule is that you can put an annotation on a variable declaration, method declaration, or class declaration. This line of code declares a variable, so we can put an annotation on it.

Next, we add a quick unit test to NumButtonControllerTest.java. We'll let you do this on your own, as you should be pretty pro at it by now.

After that, we'll go into the XML file, and put `onAction="#onPlus"` onto the plus button. Then we want to go to AppTest.java and add an integration test for the whole thing. In doing so, I found it useful to abstract out the code for "click on a sequence of buttons" (that we wrote in a former test):

```
private void clickButtonsFor(String str, FxRobot robot) {
  for (char digit : str.toCharArray()) {
    if (digit == '.') {
      robot.clickOn("#dot");
    } else {
      robot.clickOn("" + digit);
    }
  }

}
```

and then I can make use of that in my plus button test:

```
@Test
void test_plusButton(FxRobot robot) {
  clickButtonsFor("123.5", robot);
  robot.clickOn("#Enter");
  clickButtonsFor("234.25", robot);
  robot.clickOn("#plus");
  FxAssert.verifyThat("#currentNumber", TextInputControlMatchers.hasText(""));
  FxAssert.verifyThat("#rpnstack", ListViewMatchers.hasItems(1));
  FxAssert.verifyThat("#rpnstack", ListViewMatchers.hasListCell(357.75));
}
```

After writing this, I realized that (a) I want to re-use the same structure for my other buttons, and (b) I'd like to test with and without hitting Enter between the second number and the plus (or minus or whatever).

Accordingly, as soon as I wrote this, I refactored it and put it to use to test with and without enter:

```
void test_button_helper(FxRobot robot, String btnName, String inp1,
                        String inp2, double ans, boolean useEnter){
  clickButtonsFor(inp1, robot);
  robot.clickOn("#Enter");
  clickButtonsFor(inp2, robot);
  if(useEnter) {
    robot.clickOn("#Enter");
  }
  robot.clickOn(btnName);
  FxAssert.verifyThat("#currentNumber", TextInputControlMatchers.hasText(""));
  FxAssert.verifyThat("#rpnstack", ListViewMatchers.hasItems(1));
  FxAssert.verifyThat("#rpnstack", ListViewMatchers.hasListCell(ans));
}

@Test
void test_plusButton_wo_enter(FxRobot robot) {
  test_button_helper(robot, "#plus", "123.5", "234.25", 357.75, false);
}
@Test
void test_plusButton_w_enter(FxRobot robot) {
  test_button_helper(robot, "#plus", "93.7", "24.3", 118, true);
}
```

We'll let you handle minus, times, and divide. These should be pretty straightforward, as they are all quite similar to plus.

5 Error Handling

Before we proceed, take a moment to appreciate that you have a calculator that works as long as the user does not do anything invalid. First of all, what would be invalid here? The user might put something that is not a legitimate number in the text box (*e.g.* 3.....4...5 can be entered with the buttons, or they could just type anything), or they might try to add/subtract/multiply/divide when there are not enough operands available. What happens when the user does such a thing? At the moment, an exception will print out on the terminal. That exception is probably not very meaningful to most users, so we should try to provide a better solution.

Before we try to fix this problem, let us take a look at what will happen (from a technical perspective) in these situations. Any of these actions will go through one of the methods in NumButtonController (onEnter, onPlus, onMinus, onTimes, onDivide). If the problem is with the current number not being a valid double, we will get a NumberFormatException from Double.parseDouble in pushCurrentNumIfAny. If the problem is that there are not enough operands, one of the calls to myStack.remove(myStack.size() - 1) will throw an IndexOutOfBounds exception as we try to remove from index -1.

It is also worth noting that our current binOp function from RPNStack makes a weak exception guarantee. If there is one operand on the stack, we will remove it, then throw an exception on the second line. This behavior will be very annoying to the user: if they make a mistake, the top operand will be deleted. We would much rather have a strong exception guarantee! In this particular case, we can simply check if there are enough operands before we remove any, and if not throw an exception:

```
  protected void binOp(BinaryOperator<Double> op) {
    if (myStack.size() < 2) {
      throw new IllegalStateException("That operations requires 2 operands on the stack,
                                       myStack.size());
    }
    double d1 = myStack.remove(myStack.size()-1);
    double d2 = myStack.remove(myStack.size()-1);
    myStack.add(op.apply(d2, d1));

  }
```

Now, let us return to the question of displaying the error to the user in a better way. What we would like to do is display a dialog to the user. We can use JavaFX's Alert class to do so, but where should we do this? Our controller seems to be the only choice, as that is where the code is getting called. However, this is a poor choice. One reason is the philosophical separation of the view and controller. Displaying the error is a view job, and we are in the controller code. Another is that we would rather be able to test our controllers (including their error behavior) without popping up dialogs (and thus needing to go examine the dialog to determine what happened). A third is that we would be writing try/catch in each method to handle this problem.

Instead, we can use Thread.setDefaultUncaughtExceptionHandler to specify what to do with the uncaught exceptions on the JavaFX thread. In particular, we can start by writing this class:

```
public class ErrorReporter implements Thread.UncaughtExceptionHandler {

  @Override
  public void uncaughtException(Thread thread, Throwable error) {
    //put this in for debugging: error.printStackTrace();
    while(error.getCause() != null) {
      error = error.getCause();
    }
    Alert dialog = new Alert(Alert.AlertType.ERROR);
    dialog.setHeaderText(error.getClass().getName());
    dialog.setContentText(error.getMessage());
    dialog.showAndWait();
  }

}
```

This class implements Thread.UncaughtExceptionHandler, which is the interface for "what to do with uncaught exceptions on a thread". That interface specifies one method, uncaughtException, which takes the thread on which the exception happened, and the exception (which could be any Throwable).

We might still want the full stack trace (*e.g.*, for debugging), so we may wish to start with `error.printStackTrace();` (that line is commented out in this example). The next bit seems a little strange: we have a loop where we update `error` to be `error.getCause()` as long as `error.getCause()` is not null. This loop "unwraps" any wrapping of exceptions that might have happened. In particular, JavaFX calls our controller through reflection, so the exception we threw is wrapped in an InvocationTargetException. JavaFX then wraps that in a RuntimeException (presumably as it needs it in some context where the checked InvocationTargetException is problematic). So we need to unwrap back through RuntimeException → InvocationTargetException → whatever the real exception is.

After that, we create an Alert, set its title and contents, and show it. Using showAndWait prevents us from doing anything with the calculator until we dismiss the dialog.

Our error reporting is now a bit better, but still not great. We probably do not actually want to report errors with programming terminology (like IllegalStateException and NumberFormatException). The messages that come from NumberFormatException are themselves not super clear ("multiple points"). We could do a lot here to improve the error reporting, but that is not the main goal of this walkthrough—we have shown you the basic mechanics.

Before we wrap up, we need to add some test cases for our ErrorReporter. It turns out that doing this requires a couple new things. We can write this test as follows:

```
@ExtendWith(ApplicationExtension.class)
public class ErrorReporterTest {
  @Test
  public void test_alert(FxRobot robot) {
    ErrorReporter er = new ErrorReporter();
    Platform.runLater(()->er.uncaughtException(Thread.currentThread(),
                                              new IllegalStateException("Test exception
    WaitForAsyncUtils.waitForFxEvents();
    DialogPane errorDialog = robot.lookup(".dialog-pane").queryAs(DialogPane.class);
    assertEquals("java.lang.IllegalStateException", errorDialog.getHeaderText());
    assertEquals("Test exception", errorDialog.getContentText());
    Node ok = errorDialog.lookupButton(ButtonType.OK);
    assertNotNull(ok);
    robot.clickOn(ok);
  }

}
```

The first thing we do is make a new ErrorReporter (nothing new here). The next line uses Platform.runLater. As you know by now, whenever we are not on the JavaFX thread (whether in our testing code or in our application code) and we want to do something that modifies the GUI, we need to use Platform.runLater to run the code on the JavaFX Thread. As before, we need to use WaitForAsyncUtils.waitForFxEvents() to wait until JavaFX processes our request. If we did not have this call, the rest of the code might run before we do the er.uncaughtException call.

After we wait for JavaFX to process our request, we want to get our error dialog to check out its contents. We can do this by using the robot's lookup method, which takes a CSS selector. We don't have an ID, but we can use the `.dialog-pane` class, as that should be the only dialog showing in our system. We can then check that its header text and content are what we want. Finally, we can ask the error dialog to give us it OK button, check that the result is not null, then ask our robot to click that button.

We can now use this ErrorReporter to handle uncaught exceptions in our JavaFX thread by adding

```
Thread.setDefaultUncaughtExceptionHandler(new ErrorReporter());
```

to App's start method (before we do stage.show()).

6 Wrap Up

At this point, you have seen all the major components of making a JavaFX GUI: creating a view in XML, loading it with the FXMLLoader, connecting controllers to your view, connecting models to your controllers, and testing with FXRobots. You should be able to build on these ideas if you want to make a JavaFX GUI for something of your own, or if you want to build other GUIs, many of the concepts should transfer.

If you want to integrate this into a CI pipeline, you will probably want to run your tests "headless" (with no real GUI). To do so for JavaFX, you would add

```
    testRuntime 'org.testfx:openjfx-monocle:jdk-12.0.1+2'
```

to the dependencies in build.gradle, and set the system properties:

```
testfx.headless=true
testfx.robot=glass
```

If you always want to test headless, you could do this by putting

```
    systemProperty "testfx.headless", "true"
    systemProperty "testfx.robot", "glass"
```

in the test section of your build.gradle.