JavaFX notes

Prof Bill, Mar 2020

We'll learn and use JavaFX in Program #3. For FUN!

thanks...yow, bill

Sections:

1. JavaFX and VS Code
2. JavaFX basics: online learning, structure + main()
3. Panes and layout
4. Event handling
5. Goodies

Future topics: Radio buttons and checkboxes; Text input, CSS styling; Displaying images; TimeLine animation; Fancier panes: TilePane, BorderPane, GridPane

Thank you - I bow down to Noctrl's own **Dr. Godfrey Muganda** for his excellent Java textbook. I have liberally borrowed ideas from this book.
media.pearsoncmg.com/bc/abp/cs-resources/products/product.html#product,isbn=0134038177

1. JavaFX and VS Code

We did this as Homework 09,
wtkrieger.faculty.noctrl.edu/csc210-spring2020/docs/homework09.pdf

I think that these links were key for most of us.

- **Important:** This was the $$$ for most of us...JavaFX with VS Code question, stackoverflow.com/questions/54349894/javafx-11-with-vscode
- I like this Hello, world JavaFX example, github.com/openjfx/samples/blob/master/HelloFX/CLI/hellofx/HelloFX.java
- A good place to download JavaFX 11 at Gluon, gluonhq.com/products/javafx
- Reference link: OpenJFX, openjfx.io

When I create a new file and get this error, I know I need to add stuff to my launch.json.

```
Error: JavaFX runtime components are missing, and are required to run
this application
```

Related question in SO,
stackoverflow.com/questions/51478675/error-javafx-runtime-components-are-missing-and-are-required-to-run-this-appli

I don't use packages. When you do, JavaFX starts assuming a lot of things about where your code is and your libraries, etc.

When you use a new class in your code, make sure you import the JavaFX version of the class. Common things like Button and Label occur in multiple libraries!

The VS Code "Quick fix" is very helpful with all the imports we need for gui code.

2. JavaFX basics

Here we go…

Online learning

There's tons of JavaFX help for you online. Google away.

Joe K's PK 06 gave us these valuable links:

- ➔ Here's the PK PDF, wtkrieger.faculty.noctrl.edu/csc210-spring2020/docs/pk06_javafx.pdf
- ➔ Nice tutorial, www.tutorialspoint.com/javafx/index.htm
- ➔ This is a nice video series, if that works for you, www.youtube.com/playlist?list=PL6gx4Cwl9DGBzfXLWLSYVy8EbTdpGbUIG

I like examples. Lots of them.

- ➢ This guy is probably my fave, tutorials.jenkov.com/javafx/index.html
- ➢ Another good site that is chockful 'o code, www.java2s.com/Code/Java/JavaFX/CatalogJavaFX.htm

Structure, main()

Our simple apps should all have the same structure.

- You still need a main() to run.
- Your app is a subclass of the Application class in JavaFX.
- The launch() method is defined in Application.
- Inside launch(), the Application class will eventually call your start() method.

/* Quiz: Is the start() method below an example of an overload or override? */

Here's your template.

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

/**
   Simple JavaFX program displaying an empty stage.
*/
public class SimpleJavaFXApp extends Application
{
    // still need a main() to run
    public static void main(String []args)
    {
      launch(args);
    }

    // Application launch() will eventually call your start() method
    @Override
    public void start(Stage stage)
    {
        stage.setTitle("Simple JavaFX Application");
        stage.show();
    }
}
```

3. Panes and layout

This section will cover:

- ➔ Cover layout classes: VBox, HBox, and fancier Panes
- ➔ Positioning using the javafx.geometry constants: Pos.Center
- ➔ Formatting: Insets, setMargin(), setPadding()
- ➔ Nested layouts

Panes are key

The **Pane** class is key. Overview:

- ➢ Pane is a container class. It holds gui objects and determines their layout, how they are displayed.
- ➢ To show something in JavaFX, you almost always put it in a Pane.
- ➢ You'll often use a Pane superclass because they provide extra, cool functionality.
    - **Hbox** is-a Pane - the "H" in HBox stands for "horizontal"...objects are placed horizontally, left to right.
    - **VBox** is-a Pane - the "V" in VBox stands for "vertical"...objects are placed vertically, top to bottom.
    - Pane has many fancier subclasses as well: AnchorPane, BorderPane, DialogPane, FlowPane, GridPane, StackPane, TextFlow, TilePane.
    - You don't *need* these fancy Panes for Program #3. But if you want to learn, just google 'em up. There's tons of code out there about these guys.

Examples, examples, examples

I like simple HBox and VBox for our simple program. A couple examples to get you going, www.developer.com/java/data/working-with-javafx-ui-layouts.html

Oracle provides some code examples for the fancier Panes:
docs.oracle.com/javafx/2/layout/builtin_layouts.htm

Another source with fancy Pane examples, zetcode.com/gui/javafx/layoutpanes

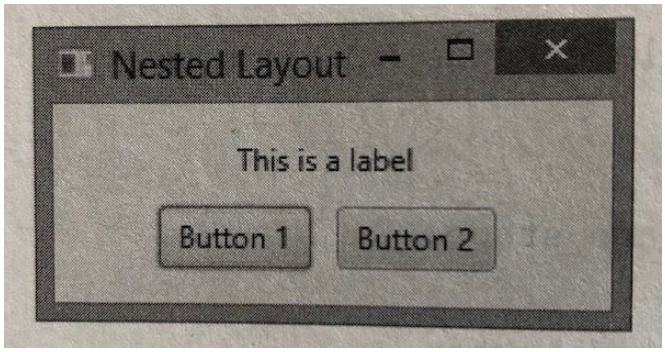
Some code snippets:

```
// create a VBox with 10 pixel spacing, center contents
VBox vb = new VBox(10);
vb.setAlignment( Pos.CENTER);

// addAll() method to add many objects to pane at once
HBox hb = new HBox();
Label lab1 = new Label( "Test case");
Button b1 = new Button( "OK");
hb.getChildren().addAll( lab1, b1);

// add() method to add objects one at a time
HBox hb2 = new HBox();
Label lab2 = new Label( "Yes another test case");
hb2.getChildren().add( lab2);
Button b2 = new Button( "Still OK");
hb2.getChildren().add( b2);

// To format Pane padding and margins, use Insets
// Source: zetcode.com/gui/javafx/layoutpanes
HBox root = new HBox(5);
root.setPadding(new Insets(10));

// Clear all objects from a Pane; google javafx clear pane
myPane.getChildren().clear();
```

It's common to nest Panes to get a desired UX: HBox within a VBox, for example. Go!

1. Let's create a VBox, v1
2. Create a Label("This is a label) and add to v1
3. Create an HBox, h1
4. Create two Buttons ("Button1" and "Button2") and add to h1.
5. Add h1 to v1

And...You should get this (turn the page):



Here's a more nested layout example,
www-acad.sheridanc.on.ca/~jollymor/prog24178/javafx3b.html

4. Event handling

This section will cover:

- ❏ EventHandler interface
- ❏ 4 options: class, inner class, anonymous class, lambda expr
- ❏ Passing information to event handlers
- ❏ Determining event target

EventHandler

EventHandler is an interface defined in JavaFX. It's generic and has one method.

Javadoc: docs.oracle.com/javase/8/javafx/api/javafx/event/EventHandler.html

It's pretty common to create your own EventHandler subclass. Example!

```
// My simple handler is a subclass of EventHandler
class SimpleHandler implements EventHandler<ActionEvent>
{
    @Override
    public void handle(ActionEvent event)
    {
        // code here to do something!
    }
}

// snippet: create a button, create a handler, use it for the button
Button b1 = new Button("Test case");
SimpleHandler handy = new SimpleHandler();
b1.setOnAction(handy);
```

---

4 options

The 4 handler options are: 1) regular, old class, 2) inner class, 3) anonymous class, and 4) lambda expression.

In the example above, SimpleHandler can be a **regular, old class**. In this case, it will reside in SimpleHandler.java. No problemo.

But this approach can get cumbersome. Here's why:

- It's common to have lots of handlers...which means lots of little class files.
- The code in a handler is usually very specific to the gui object it is handling.
- It would be convenient to have related code in the same file.
- **Important:** You can share variables with your inner class.

So, another easy option is to use an **inner class**. Just bring the class inside the gui class, and you're set. No changes.

An **anonymous class** can make your solution even smaller. It also removes the need to name every handler class.

```
// Source: taylorial.com/cs1021/EventHandlingFX.htm
Button clickMe = new Button("Click Me");
clickMe.setOnAction(new EventHandler<actionevent>() {
    @Override
    public void handle(ActionEvent event) {
        // code to handle the event here!
    }
});
```

Finally, **lambda expressions** are the shortest cut of all. Code is placed inline and doesn't need a name.

```
// Create label, button, and attach event handler to the button.
Label lab1 = new Label("0");
Button but1 = new Button("Click");

// action: every time button is pressed, increment count on label
but1.setOnAction( event -> {
    int count = Integer.parseInt(lab1.getText());
    count++;
    lab1.setText(String.valueOf(count));
});
```

Nice treatment of our 4 options here: taylorial.com/cs1021/EventHandlingFX.htm

For Program #3, **my recommendation** is inner classes. Keep it simple.

That said, lambda expressions are **red hot** these days and very common in languages like Javascript. If you're adventurous, give it a try!

Passing information to event handlers

There are two ways to pass information to your EventHandler class: lazy and ctor.

Lazy way - Of course, this is the most popular option, especially for our small gui programs. EventHandler code can access objects that are declared as class variables.

```
public class MyApp implements Application {
  private Wheel myWheel;

  public void start(Stage stage) {
    // code here…
  }

  class SimpleHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
      // code here...
      // you can access myWheel object here!
    }
  }
}
```

The lazy option works for inner classes, anonymous classes, and lambda expressions. It doesn't work if your EventHandler is a separate class (in a separate file). For larger programs, the lazy way may make your code difficult to understand and debug.

ctor way - Create a ctor for your EventHandler and pass in whatever objects you need.

```
public class MyApp implements Application {
  public void start(Stage stage) {
    // code here…
    Wheel w = new Wheel();
    SimpleHandler sh = new SimpleHandler( w);
  }

  class SimpleHandler implements EventHandler<ActionEvent> {
    private Wheel w;
    public SimpleHandler( Wheel w) {
      this.w = w;
    }
    public void handle(ActionEvent event) {
      // code here...
      // you can access this.w here
    }
  }
}
```

Determining event target

You can use one EventHandler for multiple JavaFX gui objects. In that situation, you may need to determine for which object the EventHandler was called.

The handler's event parameter knows its object, called "target".

```
class SimpleHandler implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent event) {
    // note: cast the JavaFX class, a Button in this case
    Button clickedButton = (Button) event.getTarget();
    // more code...
  }
}
```

5. Goodies

OK - this section describes JavaFX goodies until I drop. (ha)

The ordering is willy-nilly.

Alert

Alert is like a read-only dialog box that waits (in your code) until the user hits the OK button. Pretty handy!

Examples: code.makery.ch/blog/javafx-dialogs-official

ImageView

An ImageView displays an image. Add it to your Pane, and you're set.

```
// create Pane, a VBox
VBox vBox = new VBox();
vBox.setAlignment(Pos.CENTER);

// create an ImageView, add to the VBox
ImageView iv = new ImageView("tiger.jpg")
vBox.getChildren().add(iv);
```

The example is more thorough. It creates a FileInputStream object, which does proper error-checking.

Examples: tutorials.jenkov.com/javafx/imageview.html

TextArea

Need to show users some lines of text. Try TextArea. It's easy!

```
// create the area and add to a Pane
TextArea ta = new TextArea();
VBox vb = new VBox();
vb.getChildren().add( ta);
```

```
// get and set the text contents of the area
String s;
s = ta.getText();
ta.setText("Prof Bill. Huzzah!");
```

Ask me about my TextArea210 if you want an upgrade.

Examples: tutorials.jenkov.com/javafx/textarea.html

TextField

Need info from your user? Try a TextField.

Examples: tutorials.jenkov.com/javafx/textfield.html

Timeline animation

Disclaimer: Timeline is BIG and flexible. I'm only showing one tiny sliver here. To dive deeper, hop on the google train.

I have a nice, simple animation example on the k: drive. You press a button to start. The animation changes the background to a random color, N times. When complete, the background is reset, and we're done.

This is a simple TimeLine setup to do this. Note: You create the **TestHandler**, which is an EventHandler that holds the code being repeated by the Timeline.

```
// 12 steps, 500 milliseconds each
int NUM_ITERATIONS = 12;
int NUM_MILLIS = 500;

// create KeyFrame to "run handler for N millis"
Duration dur = new Duration( NUM_MILLIS);
EventHandler<ActionEvent> handler = new TestHandler();
KeyFrame kf = new KeyFrame( dur, handler);

// create Timeline to run KeyFrame N times
Timeline tl = new Timeline( kf);
tl.setCycleCount( NUM_ITERATIONS);
tl.playFromStart();
```

PieChart

Just email **Luis G**. He's using a PieChart in Program #3, and he's an expert!

Or, google javafx piechart. (smile)

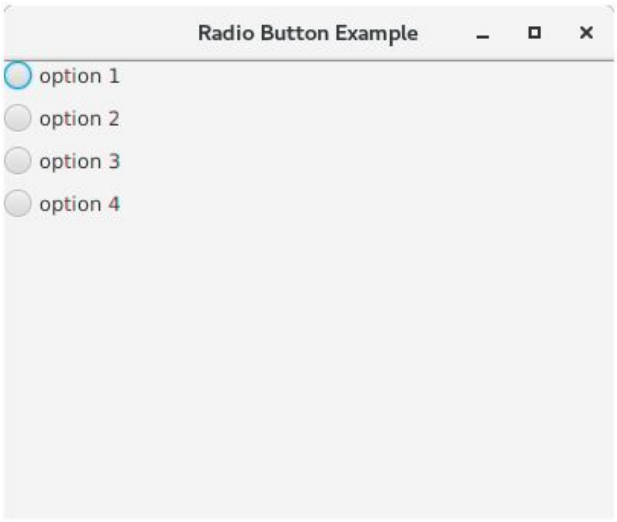
RadioButton, Checkbox

A **RadioButton** can be selected or not.

Query a RadioButton about its state with the **isSelected()** method.

Put them in a **ToggleGroup**, and then only one RadioButton can be selected at a time.

Here's a good example: tutorials.jenkov.com/javafx/radiobutton.html



Source: www.javatpoint.com/javafx-radiobutton

Checkbox

Checkbox is like RadioButton, except you can select more than one checkbox.

Source: www.javatpoint.com/javafx-checkbox

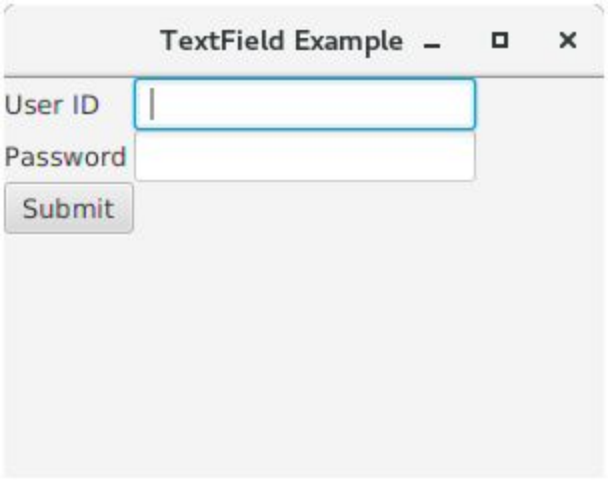
TextField

Use TextField to ask the user for data.

The getText() method returns the current text in the field.

TextFields only work with Strings. If you want an integer out of this, then…

```
// 1) get String from TextField, convert to int
int number;
try {
  // tf is a TextField obj
  number = Integer.parseInt( tf.getText());
} catch (NumberFormatException e) {
  number = 0; // zero is default
}

// 2) convert int to String and set value of TextField, tf2
int number2 = 17;
tf2.setText(String.valueOf(number2));
```



Source: www.javatpoint.com/javafx-textfield

thanks...yow, bill