COMP10062: Week 7 Guide

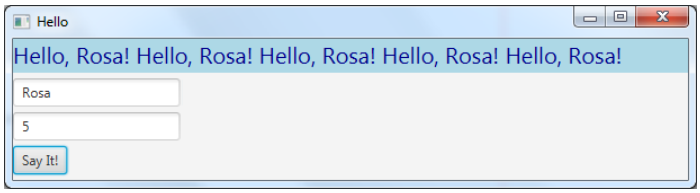
Sam Scott, Mohawk College, 2022

0 Reading for this Week

For this week, all you really need is this guide, but you could look over **sections 5.4 (GraphicsContext and Labels), 6.8 (Buttons), pp. 555-557 of section 7.6 (TextFields), and p. 649 of section 8.6 (Event Driven Programming).** Now that you know about arrays, you might also be interested in pp. 560-563 of section 7.6 on drawing Polygons.

1. Placing GUI Components

In this tutorial, you will create the layout for the "Hello" app shown below. This **Graphical User Interface** (GUI) contains 4 elements: A label, two text fields and a button.



a. Download **FXGUITemplate.Java**, change the name of the class, and make sure it runs.

b. Go to the start menu and set the size and title of the window.

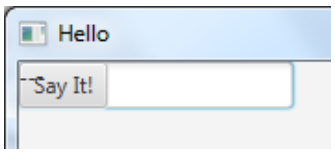
c. Under the comment that says "2. Create the GUI components", create two new text fields, a button, and a label. Here's an example of how to create a text field with some default content. See **Appendix C** at the end of this document for more.

```
TextField field1 = new TextField("Your Name Here");
```

d. Under the comment that says "3. Add components to the root", use a line like this one to add all your components to the display:

```
root.getChildren().addAll(f1, f2, b, lbl);
```

Now run the app. The elements will be placed on top of each other, something like the picture at right.



e. Under the comment that says "4. Configure the components", set the top left corner of each component using `relocate()`. Set the sizes with `setPrefWidth()`, `setPrefHeight()`, or `setPrefSize()`. Set the fonts with `setFont()`. Finally, you can set colors and other configurations with `setStyle()`. The `setStyle()` method uses CSS-style property-value pairs to configure colors, borders, etc. Here's an example of how to configure a label stored in a variable named "output".

```
output.relocate(0, 0);
output.setPrefWidth(600);
output.setFont(new Font("System", 20));
output.setStyle("-fx-background-color: lightblue;-fx-text-
fill:darkblue;");
```

Note that the CSS properties are not standard. See **Appendix C**.

2. Event Handlers

In this tutorial, you will give the "Hello" app some functionality by adding **event handlers**.

a. Create a private event handler method under the "TODO: Private Event Handlers" comment. This method will eventually be called whenever the button is pressed. (The button press is the **event**, the method **handles** the event.) Event handler methods must be void and must accept a single parameter of type `ActionEvent`.

```
private void myHandler(ActionEvent e) {
    System.out.println("Button Pressed!");
}
```

b. Now register this method as an event handler with the `setOnAction` method of the button, as shown below. Do this under the "5. Add Listeners and do final setup" comment. Then run the app and press the button. Did it work?[1]

```
myButton.setOnAction(this::myHandler);
```

Now change the method so that it reads the two text fields and uses them to change the label. It's easy to get the text from a `TextField` and set the text on a `Label` using their `getText()` and `setText()` methods. But you have to have access to these components first. This means we need to store the text fields and the label in instance variables.

c. Under "TODO: Instance Variables for View Components" comment, create private instance variables for two text fields and a label, something like this:

```
TextField nameField, numField;
Label output;
```

Then in the `start()` method, where you created, configured and placed the text fields, use these instance variables instead of the local variables you were previously using. Now you have a persistent handle on all these components.

d. Back in the event handler method, use `nameField.getText()` and `output.setText()` to say "Hello ____" where the blank is the name the user filled in.

e. Last step is to repeat the hello message n times, where n is the integer in the second text field. This can be done by concatenating repeatedly in a loop. But when you use `getText()` you get a `String`, not an `int`. Fortunately, there's a method in the `Integer` class to convert:

```
int n = Integer.parseInt(numField.getText());
```

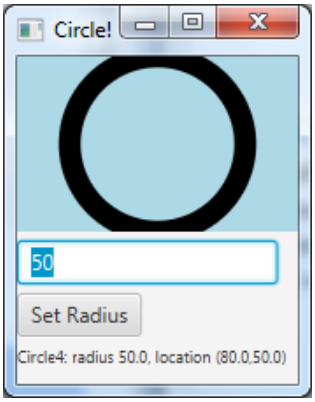
This method throws an exception if it gets a bad value, but it your program won't crash.

See **HelloWorldGUI.java** on Canvas for a version of the finished product.

---

[1] The syntax `a::b` is used to refer to method `b` from object `a`. This is a new piece of "syntactic sugar" added in Java 8. In the background, Java unpacks this expression into something called an "anonymous inner class" that implements an "interface" and calls the method you named.

3. Model vs. View



It's often useful to think of a GUI as the user's **view** of an object known as the **model**. Buttons can correspond to methods, labels can be used to display return values from methods, and text fields can correspond to parameters you would pass to the methods. In this tutorial we'll put a GUI view onto a model we've seen before: **Circle4.java**. You can get this class from Canvas.

a. Create an app from the `FXGUITemplate` that looks something like the one on the right. The four components are a `Canvas`, a `TextField`, a `Button` and a `Label`. The button has an event handler attached to it.

   A `Canvas` is a blank component that you can draw on. You create it like this:

   ```
   Canvas c = new Canvas(width, height);
   ```

   To draw on it, you get its `GraphicsContext`, like this:

   ```
   GraphicsContext gc = c.getGraphicsContext2D();
   ```

   Don't worry about making your GUI look exactly like the picture above. Just make sure you can see all the components and that the event handler is hooked up properly. Remember to store some of these components in instance variables so that you can access them from the handler.

b. In the instance variables section, declare a variable to hold the model. In this case, the model object is of type `Circle4`.

c. Under the comment "1. Create the model", create a new `Circle4` object, something like this:

   ```
   model = new Circle4(50, 80, 50);
   ```

d. It would be useful at this point to create a private helper method called "refresh" or something like that. The job of this method is to update the view to match the model. In this case, that means clear and draw the circle on the `GraphicsContext` of the `Canvas`, then update the `Label` text using `Circle4`'s `toString()` method.

   When the method is written, call it under the "5. Add Listeners and do final setup" comment.

   Try this yourself, but the finished code is in **CircleManager.java** on Canvas.

e. Now update the event handler so that it reads the new radius from the text field (use `Double.parseDouble()` to convert it), calls the `setRadius()` method of the Circle4 object, and then calls the refresh method to redraw everything.

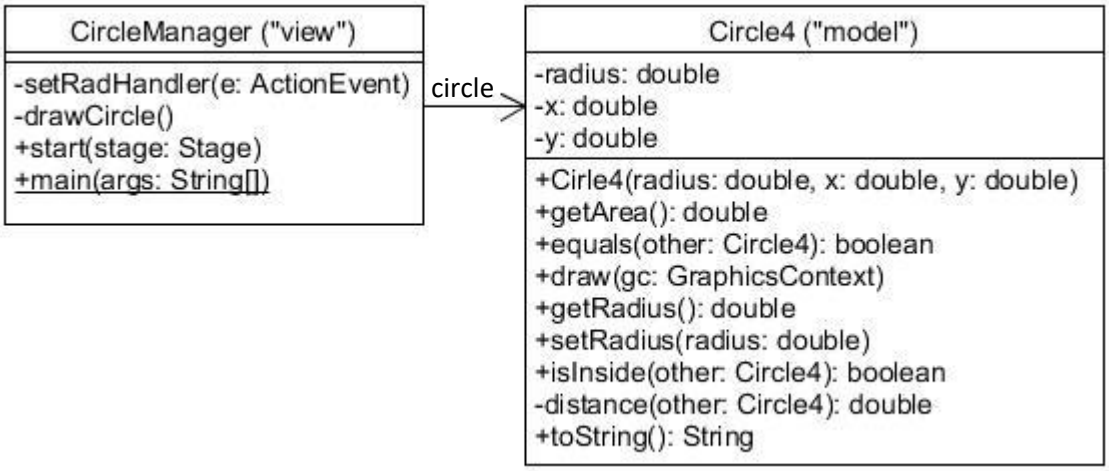
   If you feel like it, you could also call the `requestFocus()` method of the `TextField` so that the cursor returns to this field after the button is pressed.

Congratulations! You just made your first two-class GUI app, separating the view from the model. See **Appendix A** and **Appendix B** for a discussion what you just did.

Appendix A: CircleManager Class diagram

This is the class diagram for the **CircleManager.java** app on Canvas. Notice that this is an association relationship between a "view" class and a "model" class.

Your implementation might not look exactly like this one.



Appendix B: Event-Driven Programming

GUI programming is always event driven. This is a little bit different from the style of programming you're used to.

What's Different about Event-Driven Programming?

Most apps have been completely under the control of a single method – main, start, animate, etc. Once that method terminates, the app is basically finished.

Even-driven apps have a main or start method that sets things up (i.e. creates the view components and the model) but these programs continue to run after the main or start method terminates.

How Does it Work?

When the program runs, an "event handler" process starts up in the background. This process is basically an infinite loop that is constantly checking with the operating system to see if any new events have happened (e.g. a button is pressed, a key is typed, the mouse is moved, the window is closed, etc.). If it detects an event, it decides what to do about it.

If you have registered an event handler method for an event, the event handler will call that method when it detects that the event has occurred. The event handler waits for the method to finish, and then continues its infinite loop.

Gotcha: Hijacking the Event Handler

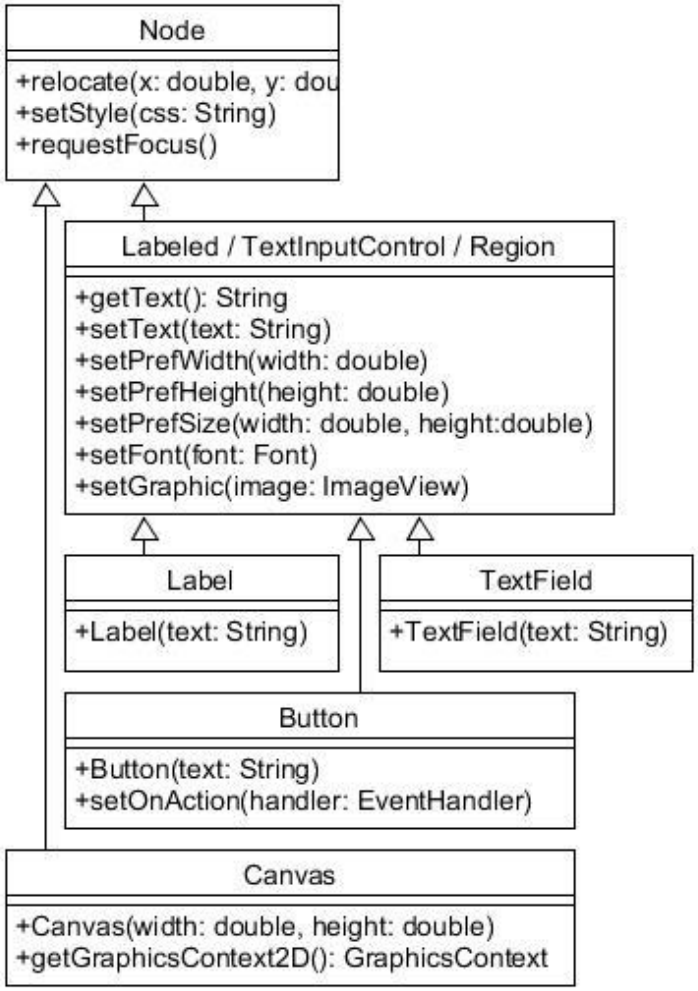
Event handlers should generally do one quick job and then exit. If they pause for too long, it will prevent the event handler from processing future events, and the window will appear to hang. Run **Hijack.java** to see an example of this.

Appendix C: GUI Components Reference

On the right is a very simplified class diagram for `Label`, `Button`, `TextField` and `Canvas`.

The arrows show how these classes "inherit" some of their methods from other classes.

For example, the `Canvas` class has the methods listed under `Canvas`, plus all the methods from `Node`. The other three component classes have their own methods plus all those from `Node` and from `Labeled / TextInputControl / Region`.



setGraphic()

This method lets you put images on buttons and labels, but it wasn't covered in the tutorials. Look at section 6.8 of the text to see how it works.

CSS Styles

The CSS styles for FX are all different from the standard web page styles, and the reference documentation is not easy to read. Here are the two you will probably need most often:

```
-fx-background-color
-fx-text-fill
```

For other possibilities, go to https://wheelercode.wordpress.com/javafx-css-properties-selectors-list/ or https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

Setting an Event Handler on a Button

```
myButton.setOnAction(this::myHandler);          ← set the handler
private void myHandler(ActionEvent e) {}        ← signature for the handler
```

Helpful Static Methods

```
Integer.parseInt(String): int                   ← convert String to int
Double.parseDouble(String): double              ← convert String to double
String.format(String, arguments…): String       ← format a string nicely
```