Chapter 11 – Graphical User Interfaces (GUI)

Contents

[1 Introduction 3](#_Toc132096129)

[2 Example: 5](#_Toc132096130)

[2.1 Hello World App 5](#_Toc132096131)

[2.2 Example: Hello World App with CSS 7](#_Toc132096132)

[3 JavaFX GUI Architecture 7](#_Toc132096133)

[4 Example: Hello World App – Modularized 8](#_Toc132096134)

[5 The *HBox* Class 10](#_Toc132096135)

[6 The *VBox* Class & *CheckBox* Class 12](#_Toc132096136)

[7 More on Modularization 13](#_Toc132096137)

[8 The *ComboBox* Class 14](#_Toc132096138)

[9 The RadioButton & ToggleButton Classes 15](#_Toc132096139)

[10 The *ListView* Class 16](#_Toc132096140)

[11 Example – More on Modularizing & Nested Panes 17](#_Toc132096141)

[12 Event Handlers 18](#_Toc132096142)

[12.1 Introduction 18](#_Toc132096143)

[12.2 Button Event Handler – Accessing a *TextField* & *TextArea* 19](#_Toc132096144)

[12.3 Handling Events on Other Controls 20](#_Toc132096145)

[12.4 Accessing a *ComboBox* in an Event Handler 21](#_Toc132096146)

[12.5 Accessing a *ListView* in an Event Handler 22](#_Toc132096147)

[12.6 Accessing a set of *CheckBoxes* in an Event Handler 23](#_Toc132096148)

[12.7 Accessing a set of *RadioButtons* in an Event Handler 24](#_Toc132096149)

[13 Maintaining State 25](#_Toc132096150)

[14 Complete Applications 26](#_Toc132096151)

[15 JavaFX Summary 28](#_Toc132096152)

[15.1 Control Summary 28](#_Toc132096153)

[15.2 Pane Summary 29](#_Toc132096154)

[15.3 Event Handler Summary 29](#_Toc132096155)

[16 Exercises 30](#_Toc132096156)

[Appendix 1 The Scene Class Hierarchy 32](#_Toc132096157)

[Appendix 2 CSS in JavaFX 33](#_Toc132096158)

[Appendix 2.1 Introduction 33](#_Toc132096159)

[Appendix 2.2 Class Styles 34](#_Toc132096160)

[Appendix 2.3 Anonymous Selectors 36](#_Toc132096161)

[Appendix 2.4 Pseudo-class Selectors 37](#_Toc132096162)

[Appendix 2.5 Descendant Selectors 37](#_Toc132096163)

[Appendix 2.6 ID Selectors 38](#_Toc132096164)

[Appendix 2.7 Practical Issues 38](#_Toc132096165)

[Appendix 2.8 Motivation for CSS 39](#_Toc132096166)

[Appendix 3 BorderPane 40](#_Toc132096167)

[Appendix 4 Slider Example 41](#_Toc132096168)

[Appendix 5 Charts 41](#_Toc132096169)

[Appendix 6 JavaFX Event Handling 42](#_Toc132096170)

[Appendix 7 Inner Classes 43](#_Toc132096171)

[Appendix 8 Event Handling using Anonymous Inner Classes 44](#_Toc132096172)

[Appendix 9 Event Handling using Lambda Expressions 45](#_Toc132096173)

[Appendix 10 Styling A GUI with Code 46](#_Toc132096174)

[Appendix 11 Resources 49](#_Toc132096175)

# Introduction

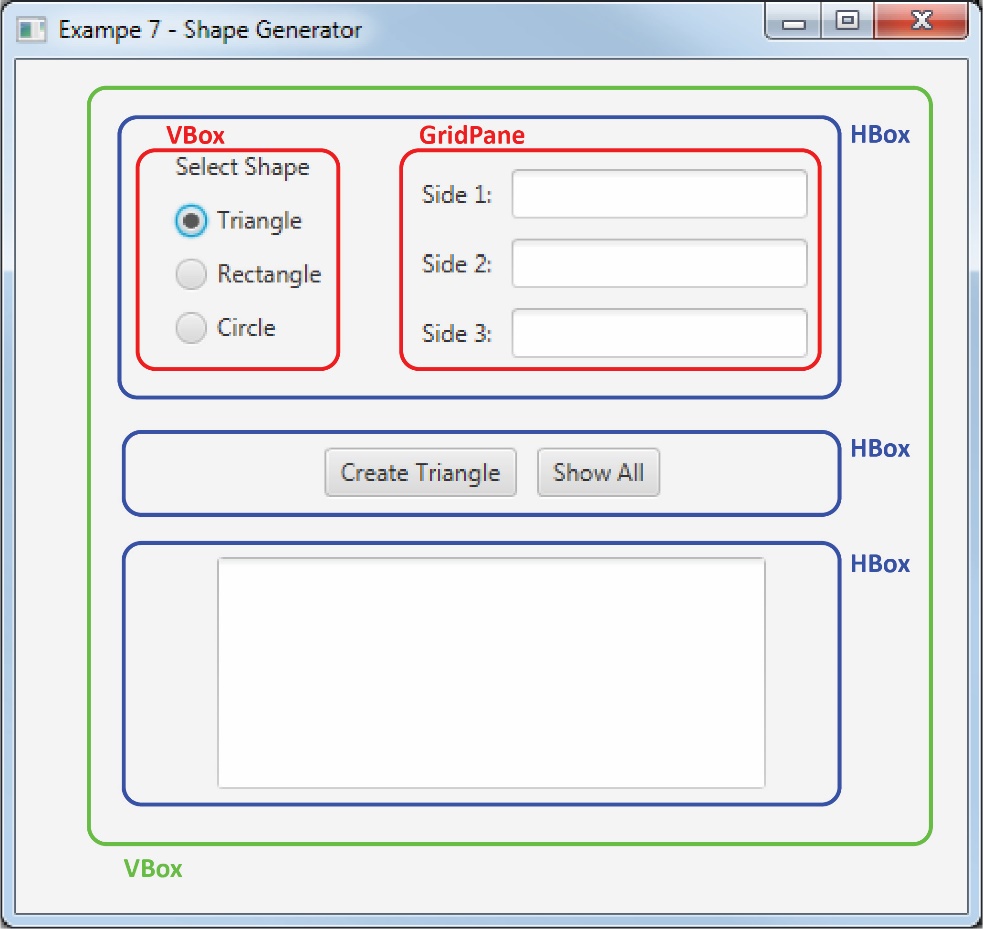
In this chapter we consider how to build GUI applications using JavaFX. To do JavaFX programming in Eclipse, see Lab 1, Sections 3-8. We start with some basic definitions:

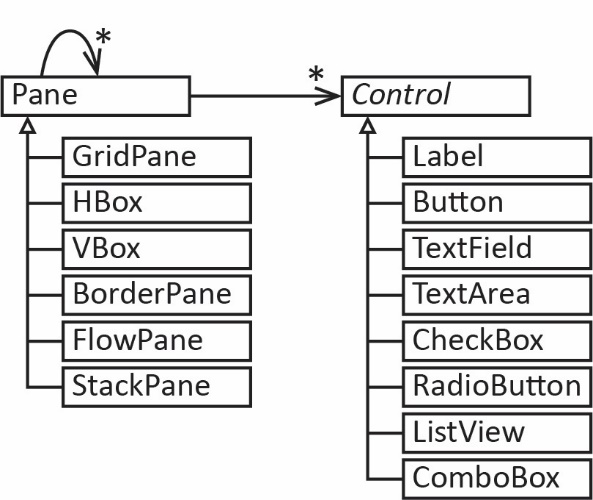
1. ***Graphical User Interface* (GUI)** – A window that allows the user to interact with graphical icons (controls). A GUI is also called a *form* or *window.* In web programming, it would be called a *page*.
2. ***Control*** – An element that is displayed on a GUI. In the example on the right, there are four controls: *Label, TextField, Button, TextArea*. Controls are objects so we can program them. *Controls* are sometimes called *widgets*.
3. ***Event*** *–* An *event* occurs (is *fired* or *sent*) when a user interacts with certain controls. For example, a button is clicked, and *ActionEvent* object fires. The programmer can write code to respond to an event.
4. ***Event Handler*** *–* A method that is called automatically when an event is fired. It will usually pull in information from GUI controls, process the information, and then display some result. In the example above, when the button is pressed a *button event handler* is called to obtain the name that was typed into the *TextField,* compose a message, and display it in the *TextArea*.
5. ***Event-Driven Programming*** *–* This refers to the process of writing GUI applications. The name comes from the fact that a GUI is generally static, it doesn’t do anything until the user interacts with it; hence, event-driven.

The two main goals of this chapter are:

* How to build a GUI.
* How to write event-handlers that respond to user interaction.

*User interface design[[1]](#footnote-1)*, often called *user experience* (UX) in another important consideration; however, we will not discuss this.

Below are a few of the basic ideas and concepts that surround GUI construction in JavaFX. We will learn the details as we move through the chapter.

* [*JavaFX*](https://docs.oracle.com/javase/8/javase-clienttechnologies.htm) refers to a set of classes that we use to build a GUI. To do JavaFX programming, you should use the e(fx)clipse plugin for Eclipse
* To build a GUI in JavaFX, you put *Controls* on a *Pane*. A *Pane* is an object that goes inside the window, which in JavaFX is a *Stage* object. I generally just think of a *Pane* as the GUI.
* A *Pane* is a container that arranges *Controls* in a particular layout. For example, the red *VBox* pane (outlined for emphasis) contains a *Label,* and 3 *RadioButton*s.
* Panes can be nested. For example, the blue *HBox* at the top contains a *VBox* and a *GridPane.*
* A GUI must have exactly one *root* pane. The root pane in the GUI on the right is the green *VBox*. Best practice is to name the root pane, *root*; however, I have not consistently done that in this chapter.
* *Control* is an abstract class in Java. Some common subclasses are shown in the diagram on the right. Some of the common controls are shown here:

<https://docs.oracle.com/javafx/2/ui_controls/overview.htm>

* *Pane* is a class in Java and is used as a container to hold *Controls* or nested *Panes*. Subclasses of *Pane* are used to layout controls in different ways. Some common subclasses are shown in the diagram on the right. Scroll through this page to see examples of different types of *Pane*s:

<http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm>

Java is on its third iteration of API’s to support event driven programming. AWT was first, followed by Swing, and the most current is JavaFX. *Swing* and *AWT* are still used. We will only consider *JavaFX*.

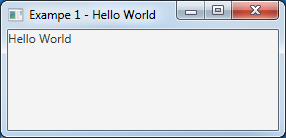
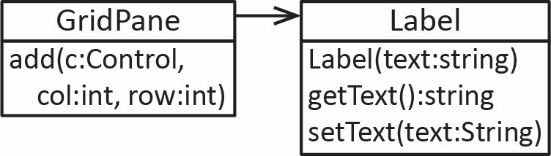
|  |  |
| --- | --- |
| AWT (1995) | <http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html> |
| Swing (1997) | <http://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html> |
| JavaFX (2008) | <https://docs.oracle.com/javafx/2/api/> |

# Example:

## Hello World App

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_1\_No\_CSS* class. Note that this and subsequent examples may differ slightly from the screen shots shown: the title will usually be different, and the default colors for the frame of the window and background might be different due to a change in operating systems.

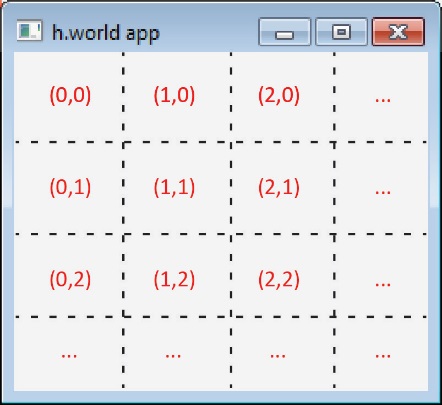
Our first example is the ubiquitous “Hello World” example as shown below, on the left. The GUI uses a *GridPane* as the *root* pane, which contains a single control, a *Label* as shown in the class diagram on the below, on the right.

A [*Label*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Label.html)is a subclass of *Control* that simply displays (non-editable) text (and optionally a graphic). One constructor accepts the text that you want to display. For example:

Label lblMsg = **new** Label("Hello World");

The *Label* class also has *getText* and *setText* methods for getting and setting the text that is displayed. We use *getText* when we consider event handlers, later. As a side note, the [*Label*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Label.html) class inherits almost 500 methods.

A [*GridPane*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/GridPane.html)is a subclass of *Pane* that holds *Controls* and arranges them in a grid with columns and rows as shown on the right. The *GridPane* only has one constructor, for example:

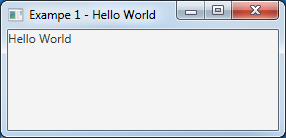
GridPane grdPane = **new** GridPane();

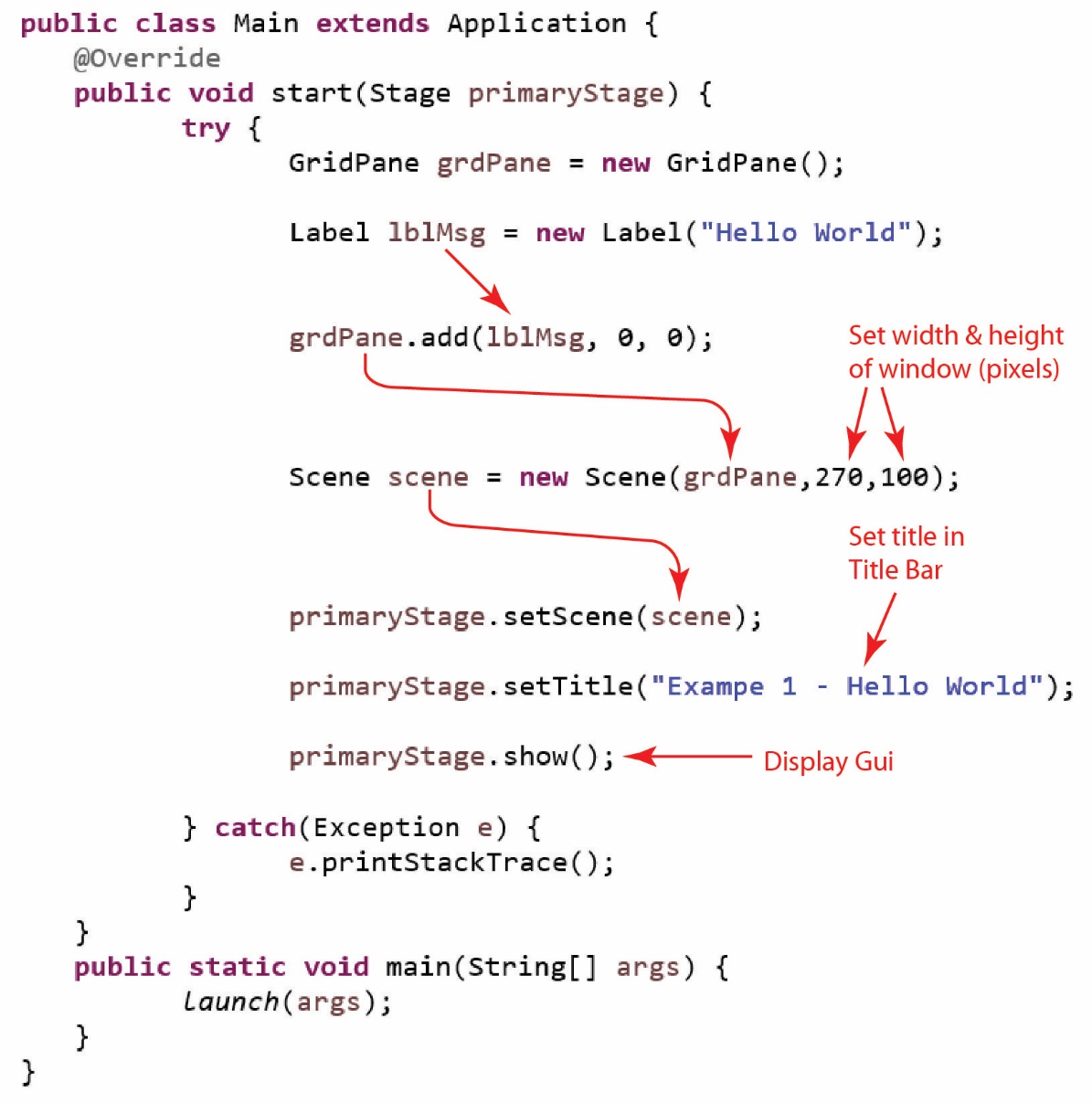
It has an *add* method that accepts the control to add and the position on the grid to put the control. For example:

Label lblMsg = **new** Label("Hello World");

grdPane.add(lblMsg, 0, 0);

adds a *Label* to the upper-left corner. Note that the second and third arguments are the *column* and *row* (zero-based).

The complete program to display the GUI is shown below. Note that *main* calls *launch* (an inherited method)and *launch* calls *start*. Thus, in a GUI application, we can think of *start* playing the role that *main* plays in a console application.



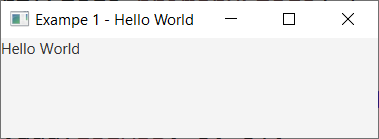
Notes:

* Although not required, there is a *try/catch* block in the *start* method in case an exception is thrown when trying to display the GUI. In Eclipse, when you create a JavaFX Project, it contains a sample GUI (an empty pane) which includes a *try/catch* in the start method.
* The sample GUI will also have an additional line of code, after the fourth line in the *try* block, as shown below, that we consider shortly.

scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

## Example: Hello World App with CSS

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_2\_CSS* class. Here, we introduce *Cascading Style Sheets* (CSS), which a useful way to style elements of a GUI. For example, things like: padding, margin, spacing, font, font size, *etc*. We only focus on the bare minimum amount of styling, typically just padding and spacing so that controls are not scrunched up on one another. We will mostly just show CSS by example. A further treatment is in an [Appendix](#Appendix_CSS_in_JavaFX).

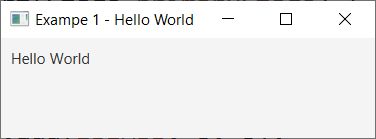
Notice the GUI considered previously (repeated on the right), that the label is displayed snuggly in the upper left corner. We can apply *padding* to the root pane by using a *Cascading Style Sheet* (*CSS*) where we write a *style definition* for the *root* pane. In the example above, the *root* pane is the *GridPane*, *grdPane.* Note: *padding* is probably what you think of as a *margin*. However, margin has a different meaning in JavaFX.

Next, let’s add a CSS to the previous example. When you create a JavaFX project in Eclipse, it creates a *CSS* file that is empty, *application.css*. We can add the *root* style definition as shown below and it will add 10 pixels of padding around the root pane (top, right, bottom, and left).

.root {

-fx-padding: 10px;

}

We add the highlighted line below to the *start* method code to apply the style definition. The result is shown on the right.

GridPane grdPane = **new** GridPane();

Label lblMsg = **new** Label("Hello World");

grdPane.add(lblMsg, 0, 0);

Scene scene = **new** Scene(grdPane,270,100);

scene.getStylesheets().add(getClass().

getResource("application.css").toExternalForm());

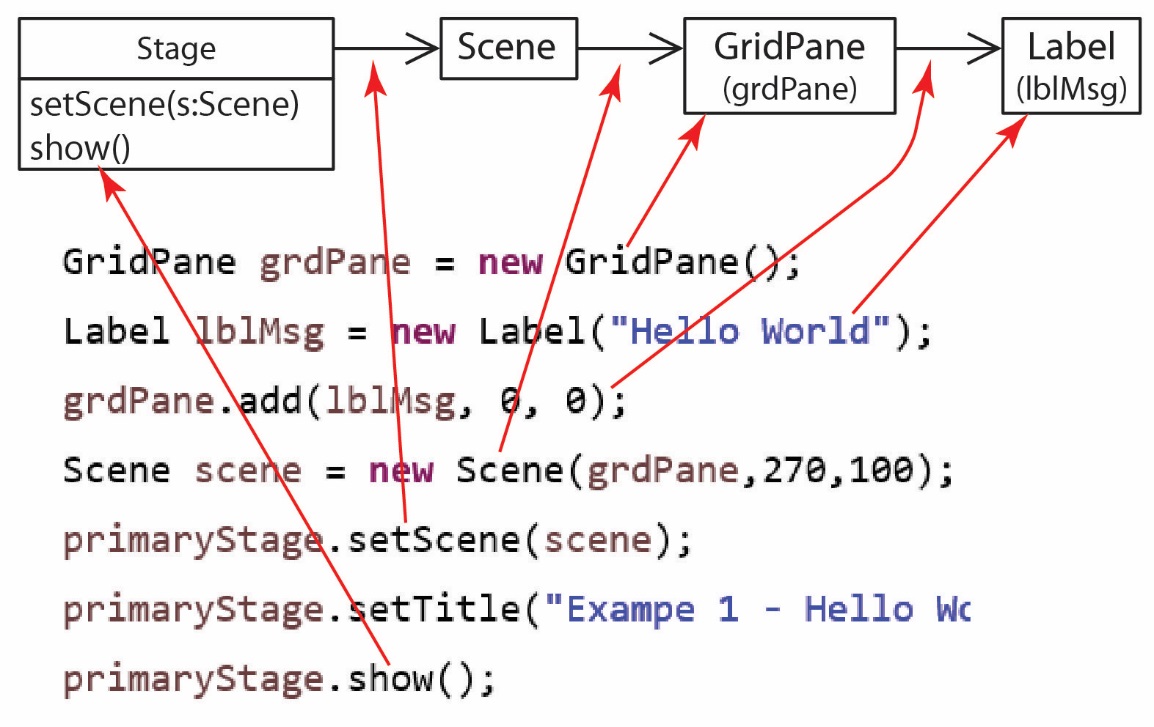
primaryStage.setScene(scene);

primaryStage.setTitle("Exampe 1 - Hello World");

primaryStage.show();

# JavaFX GUI Architecture

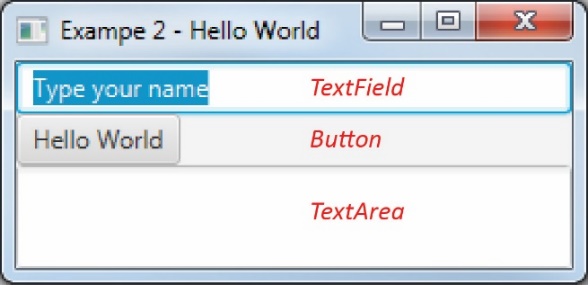
Let’s look closer at the architecture. When you run a GUI application, the *start* method is called and passed a *Stage* object. As shown in the diagram below, you put *Controls* in a *Pane*, put the *Pane* in a *Scene,* put the *Scene* in the *Stage* and then call the *show* method. In this class you will be responsible for building a *Pane* object with the required *Controls,* but the rest we will consider boilerplate code.



It is easy to create multiple *Scenes* and swap them out as needed by an application. For example, a login *Scene* has text fields for an id and password and a button that when pressed checks the credentials and then displays another *Scene,* the main page of an application. A short [video](https://www.youtube.com/watch?v=7LxWQIDOzyE) that demonstrates this. In this class, we will only consider GUI’s with a single scene.

# Example: Hello World App – Modularized

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_3\_Modularized* class.

In our next example, we will build the GUI shown on the right. First, we introduce the controls we will use.

A [*TextField*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TextField.html)is a control that a user can type a single line of text into (and/or we can set text there with code). For example, to create the *TextField* on the right:

TextField txfName = **new** TextField("Type your name");

The *TextField* class also has *getText* and *setText* methods for getting and setting the text that is displayed, which we will use when we consider event handlers.

A [*Button*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Button.html)is a control that can be pressed. We can configure the button to call an event handler when pressed; we do that in a later section. So, for now, we can display a button, and it can be pressed, but it will not produce any effect. We can create a *Button* like this:

Button btnHelloWorld = **new** Button("Hello World");

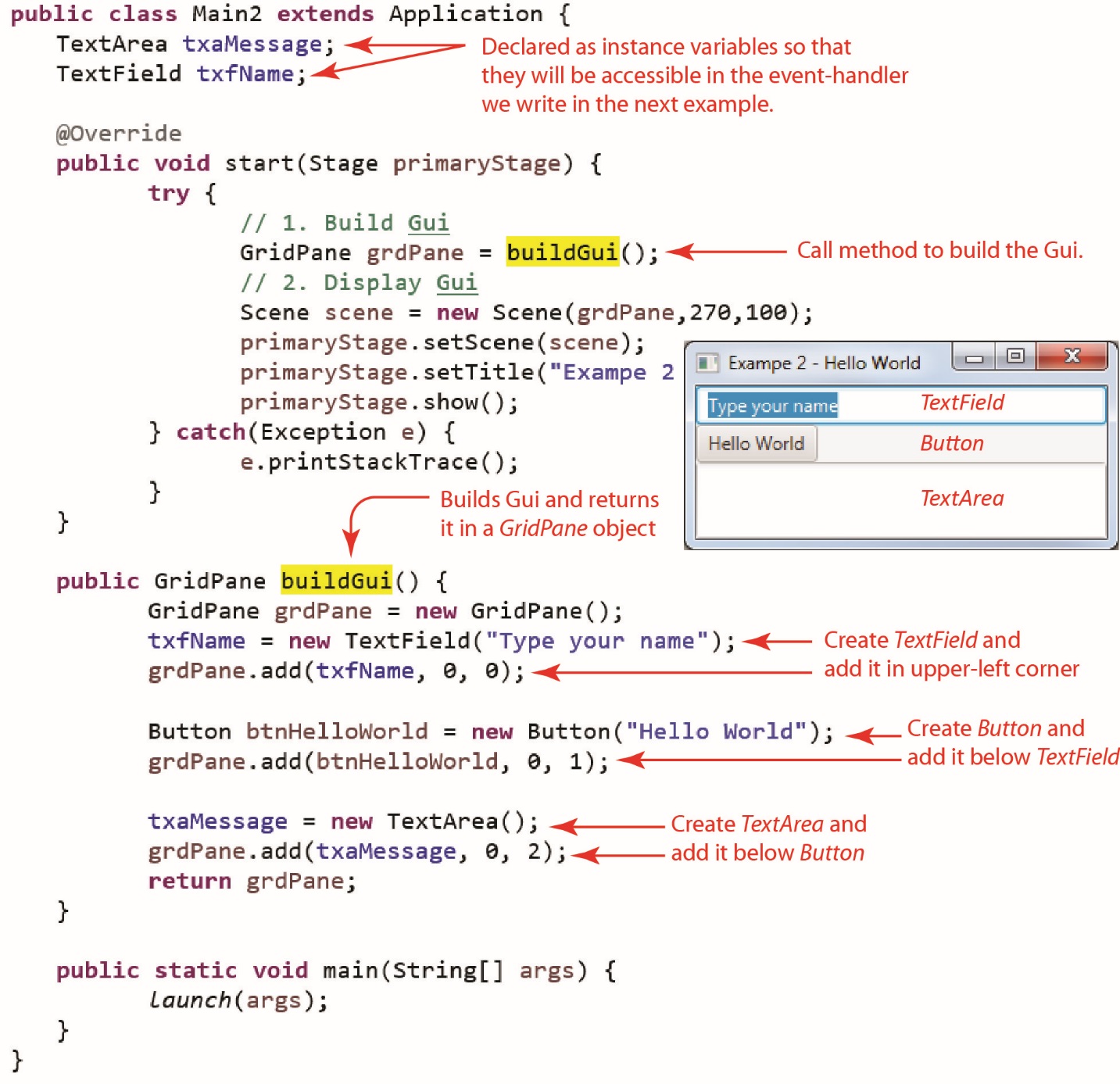
The *Button* class also has *getText* and *setText* methods for getting and setting the text that is displayed on the button itself.

A [*TextArea*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TextArea.html)is a control that can display multiple lines of text (or allow a user to type text). Frequently, we will use a *TextArea* to display the output of our code. For example:

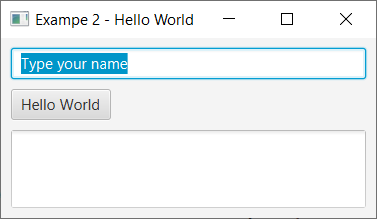
TextArea txaMessage = **new** TextArea();

The *TextArea* class also has *getText* and *setText* methods for getting and setting the text that is displayed. In some of the later examples, I may use *setPrefRowCount* and *setPrefColumnCount* to set the preferred number of rows and columns that are displayed.

Usually it takes a lot of code to build the GUI, so frequently we will *modularize* our code by writing a helper method to build and return a *Pane* object that contains the GUI. For this class, we will frequently name this method, *buildGUI.* As our GUI’s become more complex, our *buildGUI* method will use other helper methods to build the various pieces of the GUI. The program below produces the GUI shown above.

****

Notice that the *TextField* and the *TextArea* are defined as instance variables. This is important because when we define the event-handler, we will need to access these so that we can extract the name the user entered from the *TextField* and compose a message and put it in the *TextArea.* Had we defined them in the *buildGUI* method, they would have been local variables and no longer available once that method ended. Thus, they would not be accessible to the event-handler.

The example above did not have a style sheet attached; however, the ne in the code download does. As we saw previously, we can add a style definition for *root* to provide padding. However, that only applies to the edges of the pane. To create vertical distance between the cells in the *GridPane,* we add the *GridPane* style definition as shown below. There we set the *hgap* and *vgap*, which are the horizontal and vertical, respectively, distance between cells in the *GridView*. The result is shown on the right.

.root {

-fx-padding: 10px;

}

**GridPane** {

-fx-hgap:10px;

-fx-vgap:10px;

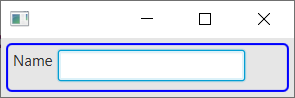
}

Note:

* In this case, we could have put the *hgap* and *vgap* in the *root* definition because the root pane, in this case, is a *GridPane*. However, the root pane could be any number of other types of pane, and these others would not have *hgap* and *vgap* properties.
* The *GridPane* style definition applies to all instances of *GridPane* in our GUI. In other words, if we had a *GridPane* inside the cell of a *GridPane*, they would both have these properties.
* The naming convention for the two style definitions is different, “.root” vs. a class name. This is explained in an [Appendix](#Appendix_CSS_in_JavaFX).

# The *HBox* Class

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_4\_Modularized\_HBox* class.

An [*HBox*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/HBox.html) is a container (*Pane*) that arranges the controls horizontally, one after the other, in a single row. For example, the code below renders as shown on the right (I have added a blue border around the *HBox* for emphasis):

Label lblPrompt = **new** Label("Name");

txfName = **new** TextField();

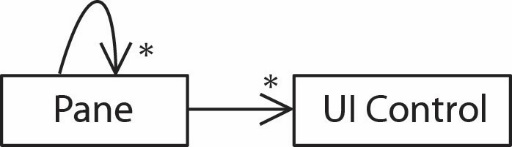
// Create HBox

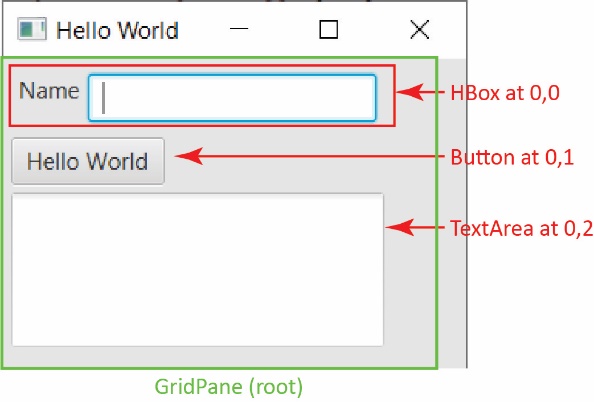
HBox hBoxName = **new** HBox();

// Add controls to HBox

hBoxName.getChildren().addAll(lblPrompt, txfName);

Note that the *HBox* has a non-obvious way to add the controls. There is not an *add* method. Instead, you call the *getChildren* method that returns a collection of the controls in the *HBox* (in this case, initially empty) and then you can call the *addAll* method to add the controls. The *getChildren* collection also has an *add* method so that you can add the controls one at a time.

One very useful concept is that we can nest panes: we can put panes inside of other panes. This provides for infinitely more layout flexibility and organization. In other words, a *Pane* can contain any number of (sub) *Panes*, and each *Pane* can contain any number of controls as shown in the class diagram on the right.

For example, consider the GUI shown on the right. Note the following:

* We use a *GridPane* as the *root* pane (green outline).
* The node at 0,0 is an *HBox* that contains a *Label* and a *TextField*.

The code below will create the GUI shown:

**public** GridPane buildGUI() {

GridPane grdPane = **new** GridPane();

// Build top row & add to GridPane

Label lblPrompt = **new** Label("Name");

txfName = **new** TextField();

HBox hBoxName = **new** HBox();

hBoxName.getChildren().addAll(lblPrompt, txfName);

grdPane.add(hBoxName, 0, 0);

// Build second row

Button btnHelloWorld = **new** Button("Hello World");

grdPane.add(btnHelloWorld, 0, 1);

// Build third row

txaMessage = **new** TextArea();

txaMessage.setPrefColumnCount(20);

txaMessage.setPrefRowCount(3);

grdPane.add(txaMessage, 0, 2);

**return** grdPane;

}

An *HBox* needs its *spacing* property set to set the distance between controls. Note: The figure above does not have this definition applied; however, the code download does.

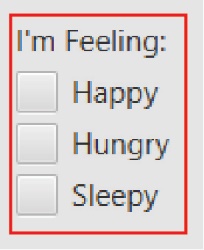
**HBox** {

-fx-spacing: 10px;

}

# The *VBox* Class & *CheckBox* Class

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_5\_CheckBoxes* class.

A [*VBox*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/VBox.html) is the same as an *HBox* except that the controls are arranged vertically, in a single column. For example, the pane on the right shows a *Label* and 3 [*CheckBoxes*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/CheckBox.html)in a *VBox* (note that I have added a red border around the *VBox* for emphasis)*.* We will discuss *CheckBox* in more detail later. The code below will produce it. First, declare as instance variables:

**protected** Label lblImFeeling;

**protected** CheckBox chkHappy, chkHungry, chkSleepy;

Helper method that will return the pane:

**private** Pane buildFeelingsEntry() {

lblImFeeling = **new** Label("I'm Feeling:");

chkHappy = **new** CheckBox("Happy");

chkHungry = **new** CheckBox("Hungry");

chkSleepy = **new** CheckBox("Sleepy");

VBox vBoxFeeling = **new** VBox();

vBoxFeeling.getChildren().addAll(lblImFeeling, chkHappy, chkHungry, chkSleepy);

**return** vBoxFeeling;

}

Notice the *buildFeelingsEntry* method uses a *VBox* but returns it as a *Pane*. Of course, we can do this because *VBox* is a subclass of *Pane*. The benefit is that if we change the *VBox* to some other type of *Pane*, then the code that calls this method will not have to change.

The *CheckBox* class has *getText* and *setText* methods. It also has an *isSelected* method that returns *true* if it is checked and *false* otherwise.

We add a style definition to our style sheet to provide spacing in the *VBox*.

**VBox** {

-fx-spacing: 10px;

}

Some examples of other types of panes, that we didn’t consider are in the *examples\_panes* package.

# More on Modularization

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_5\_CheckBoxes* class.

First, let’s summarize what we learned earlier about modularization.Instead of building the GUI in the *start* method, we will have the *start* method call a method that builds and returns the GUI.

1. First, we write a helper method, *buildGUI* that builds the entire GUI and returns a *Pane* object (which is the GUI). For example:

**private** Pane buildGUI() {

GridPane root = **new** GridPane();

...

**return** root;

}

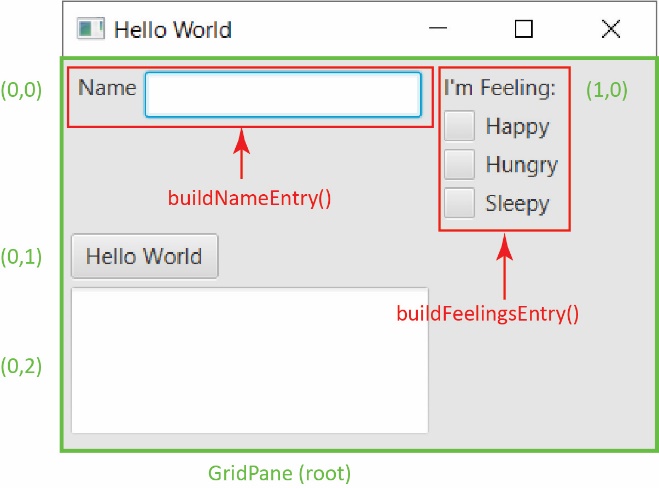
1. In *start*, we call this method and then pass the result to the *scene*.

// Create root container for controls

Pane root = buildGUI();

// Add root to Scene.

Scene scene = **new** Scene(root,400,300);

Consider the GUI on the right. We modularize further by writing helper methods to build the name entry, *buildNameEntry()* and to build the check box component, *buildFeelingsEntry()*. Then, in *buildGUI*, we call the two methods:

**private** Pane buildGUI() {

GridPane root = **new** GridPane();

Pane p = buildNameEntry();

root.add(p, 0, 0);

p = buildFeelingsEntry();

root.add(p, 1, 0);

Button btnHelloWorld = **new** Button("Hello World");

root.add(btnHelloWorld, 0, 1);

txaMessage = **new** TextArea();

txaMessage.setPrefColumnCount(7);

txaMessage.setPrefRowCount(3);

root.add(txaMessage, 0, 2);

**return** root;

}

The *buildFeelingsEntry* method was considered in the previous section. Here, we show the other helper method:

**private** Pane buildNameEntry() {

Label lbl = **new** Label("Name");

txfName = **new** TextField();

HBox hBoxName = **new** HBox();

hBoxName.getChildren().addAll(lbl, txfName);

**return** hBoxName;

}

# The *ComboBox* Class

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_6\_ComboBox* class.

A [*ComboBox*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ComboBox.html)is what we frequently call a *drop-down list.* It can be declared as an instance variable as shown below. In this case, the generic type parameter, <String> indicates that the “list” will be displayed as strings (it could hold images or other objects).

**protected** ComboBox<String> cmbSalutation;

|  |  |
| --- | --- |
|  |  |

The method below will build and return a *Pane* that contains a label and a *ComboBox* as shown on the right.

**private** Pane buildSalutation() {

VBox saluation = **new** VBox();

Label lblSalutation = **new** Label("Salutation");

saluation.getChildren().add(lblSalutation);

cmbSalutation = **new** ComboBox<>();

cmbSalutation.getItems().addAll("Mrs", "Ms", "Mr", "Dr");

cmbSalutation.setValue("Ms");

saluation.getChildren().add(cmbSalutation);

**return** saluation;

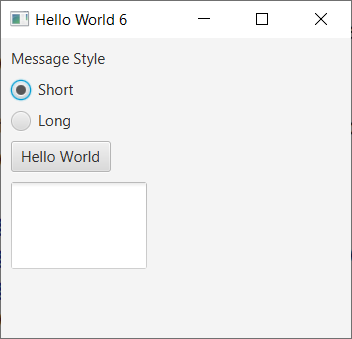
}

Notes:

* The first highlighted line adds the items to display in the *ComboBox*.
* The second highlighted line sets the item that is initially displayed.

# The RadioButton & ToggleButton Classes

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_7\_RadioButtons* class.

A portion of a *GUI* shown on the right shows a *Label* and two [*RadioButtons*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/RadioButton.html)*.* A related group of *RadioButtons* should have the property that only one can be selected. For example, if the user, for example selects “Long” (figure on the right), then “Short” will automatically be deselected.

To achieve this, the we use a [*ToggleGroup*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ToggleGroup.html)which is a container that holds instances of *RadioButton* (actually, it holds anything that is a *Toggle*)*.* The *ToggleGroup* should be declared as an instance variable. However, many times the *RadioButtons* themselves can be declared as local variables. This will be explained when we consider event handling. Here, I declare them as instance variables:

**protected** ToggleGroup tGrpStyleChoice;

**protected** RadioButton rbShort, rbLong;

The method below will build and return a *Pane* that contains a label and a two *RadioButtons* as shown above.

**private** Pane buildMessageStyleChoice() {

// Build message style component

VBox vbox = **new** VBox();

Label lbl = **new** Label("Message Style");

vbox.getChildren().add(lbl);

tGrpStyleChoice = **new** ToggleGroup();

rbShort = **new** RadioButton("Short");

rbShort.setToggleGroup(tGrpStyleChoice);

rbShort.setSelected(**true**);

vbox.getChildren().add(rbShort);

rbLong = **new** RadioButton("Long");

rbLong.setToggleGroup(tGrpStyleChoice);

vbox.getChildren().add(rbLong);

**return** vbox;

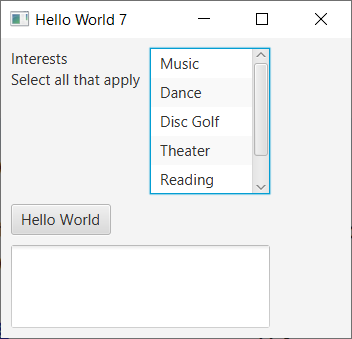
}

Notes:

* The first highlighted line creates a *RadioButton* which displays the text: “Short”.
* The second highlighted line assigns the radio button to the *ToggleGroup*
* The third highlighted line selects this radio button to be selected initially.

# The *ListView* Class

The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_8\_ListView* class.

A [*ListView*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ListView.html)displays a list of items vertically and allows for single selection, or for multiple selections. Use the declaration below as an instance variable.

**protected** ListView<String> lvwInterests = **new** ListView<>();

Note that the *ListView* class is generic; thus, we should specify the type of items the list will contain.

The method below will build and return a *Pane* that contains a label (that contains a line break) and a *ListView* as shown on the right.

**private** Pane buildInterestsEntry() {

lvwInterests.getSelectionModel().setSelectionMode(SelectionMode.***MULTIPLE***);

lvwInterests.getItems().addAll("Music", "Dance", "Disc Golf", "Theater",

"Reading");

lvwInterests.setPrefHeight(150);

lvwInterests.setPrefWidth(120);

HBox hBox = **new** HBox();

hBox.getChildren().add(**new** Label("Interests\nSelect all that apply"));

hBox.getChildren().add(lvwInterests);

**return** hBox;

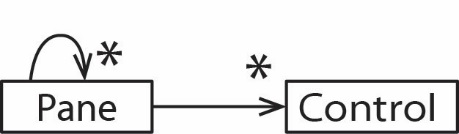
}

Notes:

* The first highlighted line sets the *ListView* so that multiple items can be selected. By default, the *ListView* only allows single selection.
* The second highlighted line assigns the items in the list.

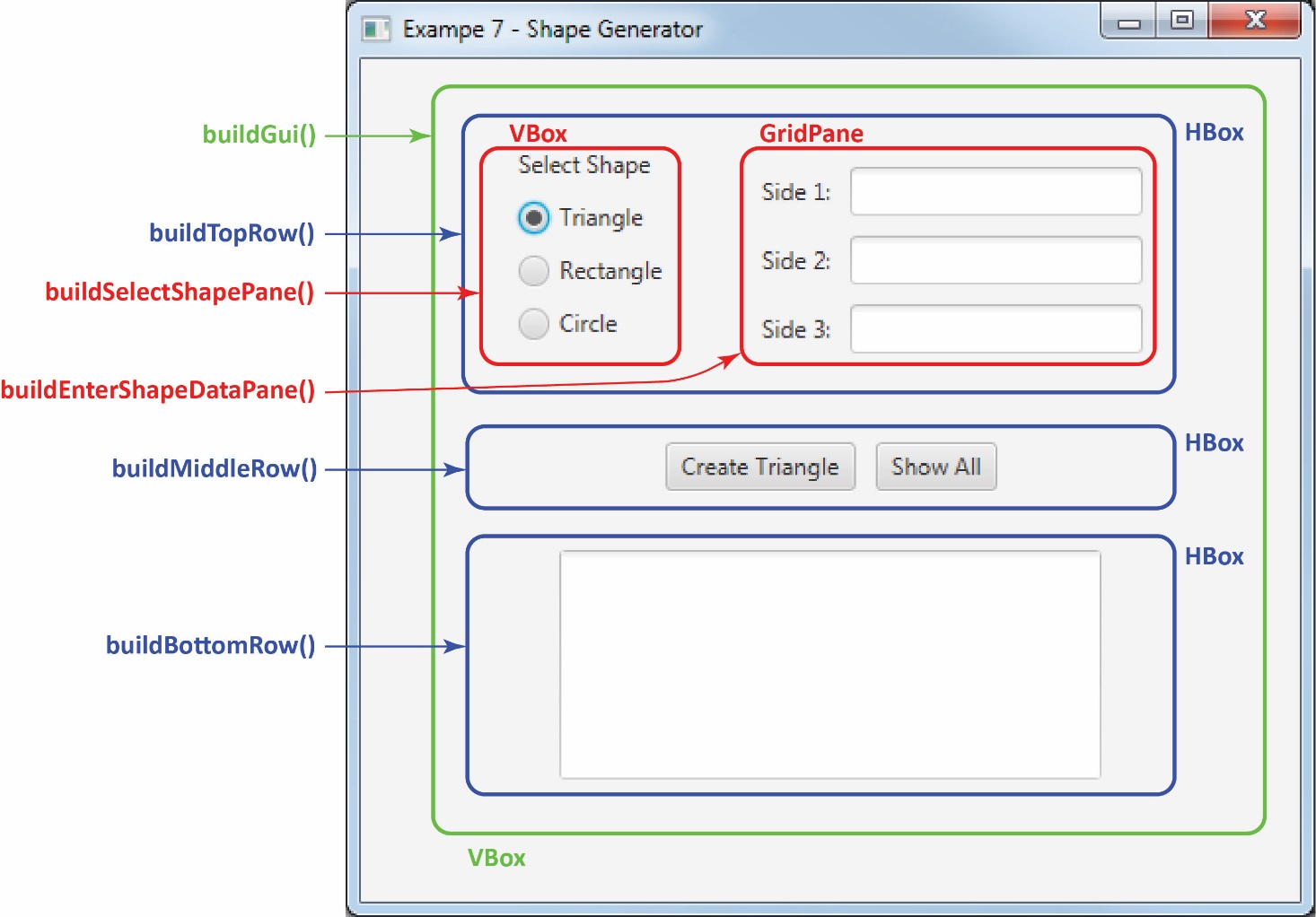
# Example – More on Modularizing & Nested Panes

The example in this section is in the *example\_shape\_generator* package.

As noted previously, panes can be nested. This is particularly useful when building GUI’s with lots of controls. As a beginner (myself included), I recommend using just: *GridPane, HBox, and VBox.* When you need to build a GUI, I recommend sketching it quickly on paper and then identifying related regions of the GUI, deciding what pane to use for that region, and then writing a method to build that region.

Consider the GUI below:

* The root pane is a *VBox* (shown in green) and is built with the *buildGUI* method.
* The top row of the root pane is an *HBox* (shown in blue). It contains two other panes: a *VBox* containing the radio buttons, and a *GridPane* containing the labels and textfields.
* The middle row of the root pane contains an *HBox* with two buttons.
* The bottom row of the root pane contains an *HBox* with a text area.
* The methods to build the various panes are shown along the left.



A portion of the code is below:

**private** Pane buildGUI(Stage stage) {

VBox root = **new** VBox();

root.getChildren().addAll(buildTopRow(), buildMiddleRow(), buildBottomRow());

**return** root;

}

**private** Pane buildTopRow() {

HBox topRow = **new** HBox();

topRow.getChildren().addAll(buildSelectShapePane(), buildEnterShapeDataPane());

**return** topRow;

}

**private** Pane buildSelectShapePane() {

VBox vBoxShape = **new** VBox();

Label lblSelectShape = **new** Label("Select Shape");

...

}

**private** Pane buildEnterShapeDataPane() {

GridPane gridShapeLengths = **new** GridPane();

lbl1 = **new** Label("Side 1:");

gridShapeLengths.add(lbl1, 0, 1);

...

}

**private** Pane buildMiddleRow() {

HBox hBoxButtons = **new** HBox();

btnCreateShape = **new** Button("Create Triangle");

hBoxButtons.getChildren().add(btnCreateShape);

...

}

**private** Pane buildBottomRow() {

HBox hBoxMessage = **new** HBox();

txaMessage = **new** TextArea();

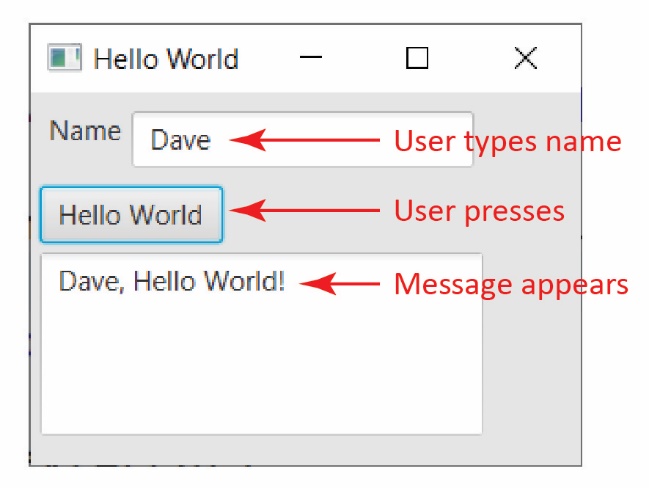
...

}

# Event Handlers

## Introduction

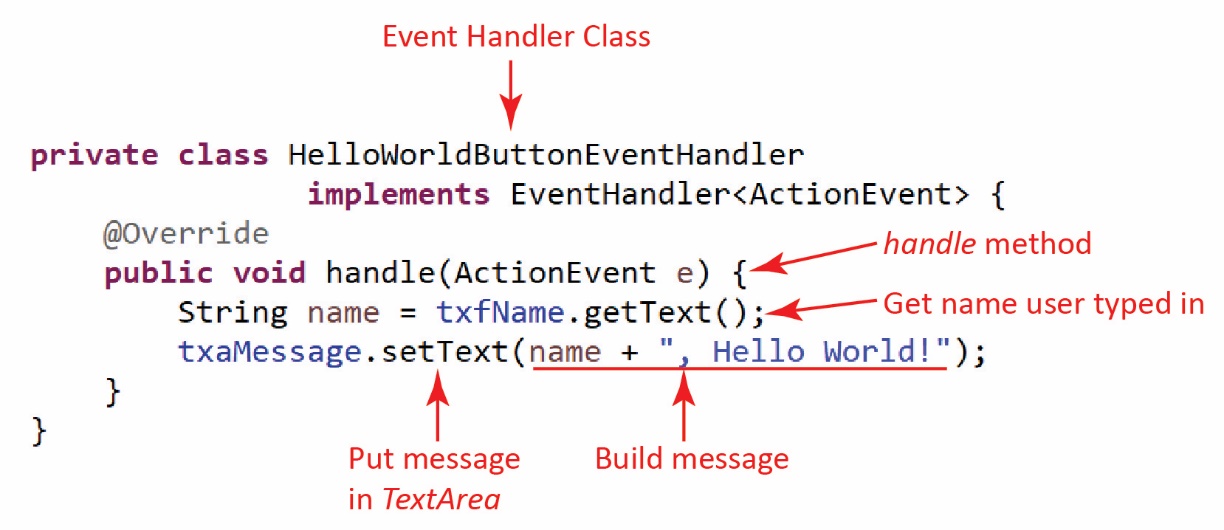
The example in this section is in the *examples\_hello\_world* package, *HelloWorld\_9\_Event\_Handling* class.

An *event handler* is code that is run when the user interacts with a control on a GUI and if the control is configured to respond to an interaction. For example, when a button is pressed in the example on the right, the event handler retrieves the name that was typed in, composes a message, and then displays it.

## Button Event Handler – Accessing a *TextField* & *TextArea*

Assuming the GUI above is written, we define an *inner class event handler.* There are other techniques such as: anonymous event handlers, lambda expression, or simply a class that has an event handler as is used in MVC (Model-View-Controller)[[2]](#footnote-2). To implement an inner class event handler for the button on the GUI above, we do the following:

1. Write a class that implements the *EventHandler* interface, which requires a *handle* method as shown below. There, we simply write code to respond to the event. However, this class is inside the main GUI class, placed at the same level as any other member of a class. Thus, we say that it is an *inner class.* An inner class can use all members of the enclosing class (even private).



1. *Register* the event handler with a control (e.g. button) that the user will interact with. When the program is run, and the button is pressed, the *handle* method is called.

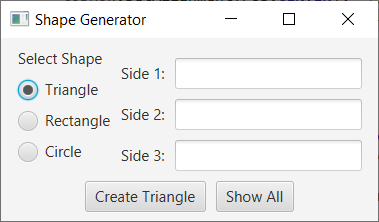
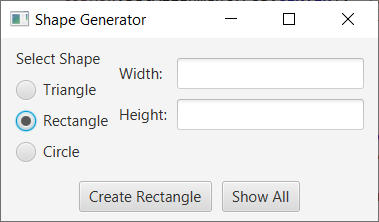
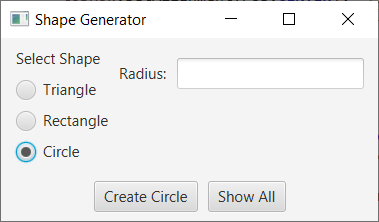


The *TextField* and *TextArea* must be declared as instance variables so that they can be accessed in the event handler. Note that the event handler does not use the *Label* nor the *Button*, so they could have been declared as local variables in the *buildGui* method. It might be useful to declare all controls as instance variables to avoid confusion.



## Handling Events on Other Controls

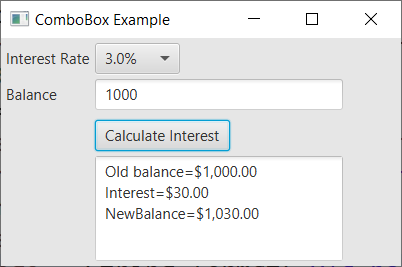
In this class we will only consider an *ActionEvent* that occurs on a *Button*. However, several other controls have a *setOnAction* method that accepts an *EventHandler<ActionEvent>* instance: *CheckBox, RadioButton, TextField, ComboBox*. The example in this [section](#_Example_–_More) has a set of radio buttons (as shown below). When one is selected, it changes the text on the “Create” button, and changes the number of input text boxes and their labels.

Again, we will only consider event-handlers for a *Button.* Other types of event handlers are considered briefly in an [appendix](#Appendix_JavaFX_Event_Handling).

## Accessing a *ComboBox* in an Event Handler

The example in this section is in the *examples\_controls* package, *ComboBoxExample* class.

Consider the GUI on the right. The user selects an interest rate from the *ComboBox,* enters a balance, presses the button then a message is displayed showing the new balance after the interest is applied.

Thus, the button’s event handler needs to access the *selected value* from the *ComboBox* as well as the value in the *TextField*. Thus, these two must be declared as instance variables.

The selected value of the *ComboBox* is obtained through its *getValue* method. For example, if the *ComboBox’s* name is *cmbInterestRate*, then we can retrieve this value with this line of code in the event handler:

String strIntRate = cmbInterestRate.getValue();

Note, however, in the example above, that *strInterestRate=”3.0%”*. Thus, we need to strip off the “%”:

strIntRate = strIntRate.substring(0,strIntRate.length()-1);

And, we need to convert it to a double (and divide by 100 to make it a decimal) before doing computations with it.

**double** intRate = (Double.*parseDouble*(strIntRate))/100.0;

The event-handler is shown below.

**private** **class** CalculateInterestEventHandler **implements** EventHandler<ActionEvent> {

@Override

**public** **void** handle(ActionEvent event) {

String strIntRate = cmbInterestRate.getValue();

// Remove "%" from the end

strIntRate = strIntRate.substring(0,strIntRate.length()-1);

**double** intRate = (Double.*parseDouble*(strIntRate))/100.0;

**double** balance = Double.*parseDouble*(txfBalance.getText());

**double** interest = balance\*intRate;

**double** newBalance = balance + interest;

String message = String.*format*("Old balance=$%,.2f\n" +

"Interest=$%,.2f\nNewBalance=$%,.2f", balance, interest, newBalance);

txaMessage.setText(message);

}

}

Register the event handler with the button by adding this line of code directly below the creation of the button in *buildGui*.

btnCalcInterest.setOnAction(**new** CalculateInterestEventHandler());

Finally, the *buildGui* method is partially shown below:

**private** Pane buildGui() {

...

Button btnCalcInterest = **new** Button("Calculate Interest");

btnCalcInterest.setOnAction(**new** CalculateInterestEventHandler());

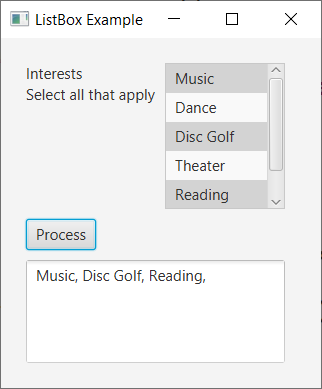
grid.add(btnCalcInterest, 1, 3);

...

}

## Accessing a *ListView* in an Event Handler

The example in this section is in the *examples\_controls* package, *ListViewExample* class.

Consider the GUI on the right. The user selects any number of items from the *ListView,* presses the button then a message is displayed showing the items that were selected.

The selected items in the *ListView* are obtained with this line of code:

List<String> allItems =

lvwInterests.getSelectionModel().getSelectedItems();

Then, we can loop over the list to access each item:

String interests = "";

**for**(String interest : allItems) {

interests += interest + ", ";

}

The event-handler is shown below:

**private** **class** ProcessEventHandler **implements** EventHandler<ActionEvent> {

@Override

**public** **void** handle(ActionEvent event) {

String interests = "";

List<String> allItems = lvwInterests.getSelectionModel().getSelectedItems();

**for**(String interest : allItems) {

interests += interest + ", ";

}

txaMessage.setText(interests);

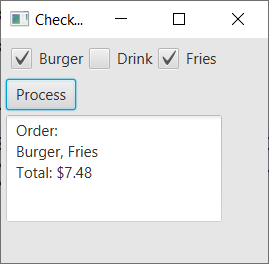
}

}

As usual, register the event handler with the button:

btnProcess.setOnAction(**new** ProcessEventHandler());

## Accessing a set of *CheckBoxes* in an Event Handler

The example in this section is in the *examples\_controls* package, *CheckBoxExample* class.

Consider the GUI on the right. The user selects any number of check boxes*,* presses the button then a message is displayed showing the items that were selected.

Note that we have individually named each *CheckBox* in a helper method to build the *HBox* that holds them.

**private** Pane buildFoodSelection() {

ckbBurger = **new** CheckBox("Burger");

ckbDrink = **new** CheckBox("Drink");

ckbFries = **new** CheckBox("Fries");

HBox hbxFood = **new** HBox();

hbxFood.getStyleClass().add("h\_or\_v\_box");

hbxFood.getChildren().addAll(ckbBurger,ckbDrink,ckbFries);

**return** hbxFood;

}

In the *handle* event handler, we simply check the *isSelected* property of each *CheckBox*:

**if**(ckbBurger.isSelected()) {

cost += 5.99;

order += "Burger";

}

**if**(ckbDrink.isSelected()) {

cost += 1.99;

order += ", Drink";

}

**if**(ckbFries.isSelected()) {

cost += 1.49;

order += ", Fries";

}

The event-handler is shown below:

**private** **class** ProcessEventHandler **implements** EventHandler<ActionEvent> {

@Override

**public** **void** handle(ActionEvent event) {

**double** cost = 0.0;

String order = "Order:\n";

**if**(ckbBurger.isSelected()) {

cost += 5.99;

order += "Burger";

}

**if**(ckbDrink.isSelected()) {

cost += 1.99;

order += ", Drink";

}

**if**(ckbFries.isSelected()) {

cost += 1.49;

order += ", Fries";

}

String totCost = String.*format*("\nTotal: $%,.2f\n", cost);

order += totCost;

txaMessage.setText(order);

}

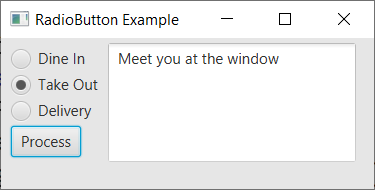
}

Register the event handler with the button:

btnProcess.setOnAction(**new** ProcessEventHandler());

## Accessing a set of *RadioButtons* in an Event Handler

The example in this section is in the *examples\_controls* package, *RadioButtonExample* class.

Consider the GUI on the right. The user selects a *RadioButton,* presses the button then a message is displayed showing the items that were selected.

Remember that we need to associate each *RadioButton* with a *ToggleGroup* to force them to work as a group (only one *RadioButton* can be selected). This is shown below in the helper method to build the *VBox* that holds them (and the *Button*). As we will see, the event handler only needs access to the *ToggleGroup* to determine which one is selected.

**private** Pane buildDiningChoice() {

tGrpDiningChoice = **new** ToggleGroup();

rbDineIn = **new** RadioButton("Dine In");

rbDineIn.setSelected(**true**);

rbDineIn.setToggleGroup(tGrpDiningChoice);

rbTakeOut = **new** RadioButton("Take Out");

rbTakeOut.setToggleGroup(tGrpDiningChoice);

rbDelivery = **new** RadioButton("Delivery");

rbDelivery.setToggleGroup(tGrpDiningChoice);

Button btnProcess = **new** Button("Process");

btnProcess.setOnAction(**new** ProcessEventHandler());

VBox vbxDiningChoice = **new** VBox();

vbxDiningChoice.getStyleClass().add("h\_or\_v\_box");

vbxDiningChoice.getChildren().addAll(rbDineIn,rbTakeOut,rbDelivery,btnProcess);

**return** vbxDiningChoice;

}

In the *handle* event handler, we use the *getSelectedToggle* methodof the *ToggleGroup* to return the *RadioButton* that was selected.

RadioButton rad = (RadioButton)tGrpDiningChoice.getSelectedToggle();

Next, we get the text of the radio button:

String choice = rad.getText();

Then, we use that text to determine which radio button was selected:

String message = "";

**switch**(choice) {

**case** "Dine In" : message = "Glad you are dining in with us";

**break**;

**case** "Take Out" : message = "Meet you at the window";

**break**;

**case** "Delivery" : message = "We will have it there shortly";

}

txaMessage.setText(message);

The event-handler is shown below:

**private** **class** ProcessEventHandler **implements** EventHandler<ActionEvent> {

@Override

**public** **void** handle(ActionEvent event) {

RadioButton rad = (RadioButton)tGrpDiningChoice.getSelectedToggle();

String choice = rad.getText();

String message = "";

**switch**(choice) {

**case** "Dine In" : message = "Glad you are dining in with us";

**break**;

**case** "Take Out" : message = "Meet you at the window";

**break**;

**case** "Delivery" : message = "We will have it there shortly";

}

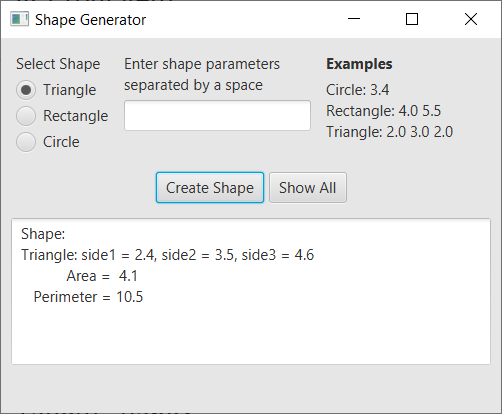
txaMessage.setText(message);

}

}

# Maintaining State

The example in this section is in the *examples\_complete\_applications\_shape\_generator\_basic* package.

Maintaining state refers to “remembering” data between events. For example, an app to order food would need to remember the orders. In a real system, you would usually use a database. However, we can also keep the data in memory. And, this is simple: use instance variables in the Gui class. For example, you might have a list to hold orders. Actually, in practice, we separate the data from the Gui class. But, for our purposes, we will just introduce instance variables as needed.

Consider the *ShapeGenerator* app on the right. You can repeatedly create shapes and the system saves them. When *Show All* is pressed, all the shapes that have been created are displayed. To accomplish this:

1. This instance variable is added to the class:

**protected** ArrayList<GeometricObject> shapes = **new** ArrayList<>();

1. And, every time we create a shape in the event-handler, we added it to *shapes*:

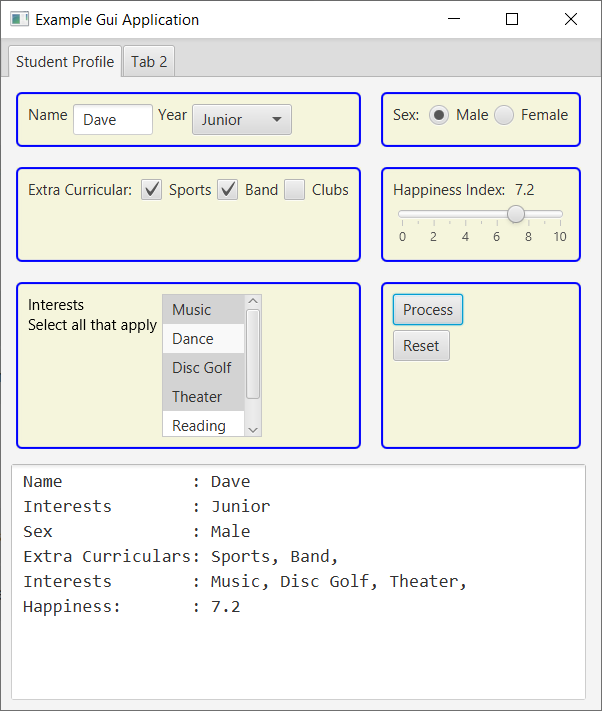
// Add to collection

shapes.add(shape);

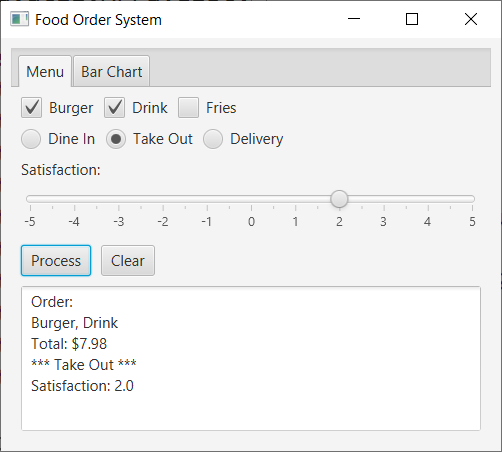
# Complete Applications

These are complete examples that can be used for reference.

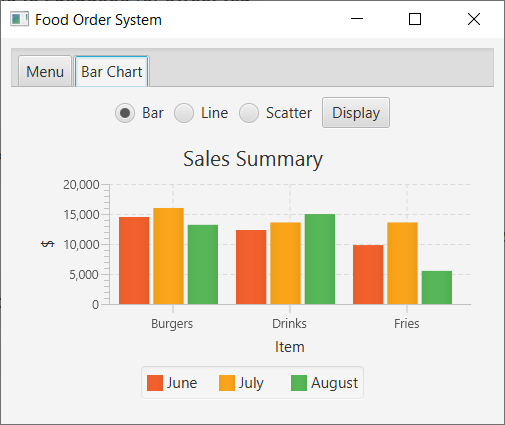
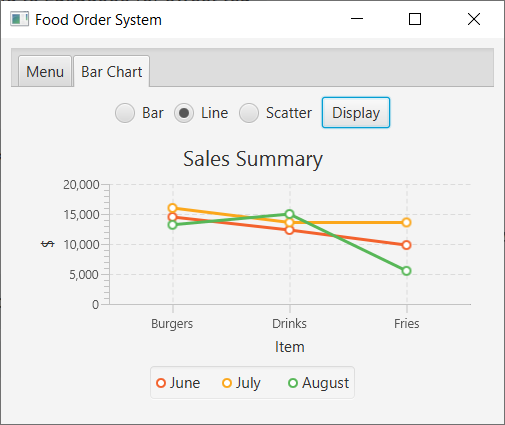
1. This example is in the *examples\_complete\_applications\_student\_data* package. It does not maintain state.

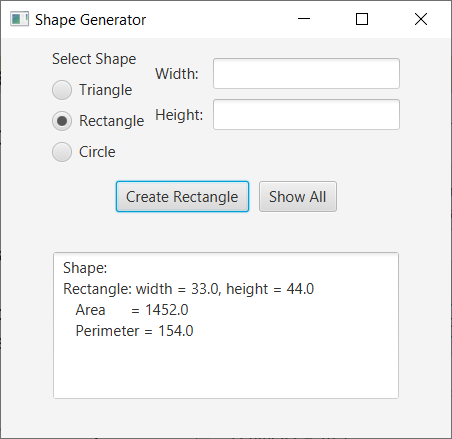
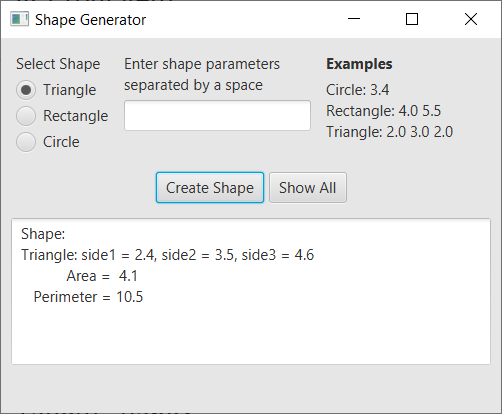


1. This example is in the *examples\_complete\_applications\_order\_menu* package. It does not maintain state.



The charts are hard coded with data, just for illustrative purposes.

1. This example is in the *examples\_complete\_applications\_shape\_generator* package. It does maintain state. Has an event handler for the radio buttons so that it changes the labels and number of text fields appropriate for the shape being created. A simpler version is below
2. This example is in the *examples\_complete\_applications\_shape\_generator\_basic* package. It does maintain state. Simpler than previous app. However, higher chance of entering bad data.

# JavaFX Summary

## Control Summary

1. Label – getText, setText

Label lbl = new Label("description")

String desc = lbl.getText()

lbl.setText(“description”)

1. TextField – getText, setText

TextField txf = new TextField("Type your name")

Str str = txf.getText() // might have to parseDouble/Int

txf.setText(“stuff”)

1. Button – getText, setText

Button btn = new Button("description")

String str = btn.getText(“description”)

btn.setText(“description”)

1. TextArea – getText, setText

TextArea txa = new TextArea()

String str = txa.getText()

txa.setText(“stuff \n more stuff”)

1. CheckBox – getText, setText, isSelected, setSelected

**protected** CheckBox ckbBurger, ckbDrink, ckbFries;

**if**(ckbBurger.isSelected()) {

...

}

**if**(ckbDrink.isSelected()) {

...

}

...

1. RadioButton – getText, setText, isSelected, setSelected, setToggleGroup

**protected** RadioButton rbDineIn, rbTakeOut, rbDelivery;

**protected** ToggleGroup tGrpDiningChoice;

tGrpDiningChoice = **new** ToggleGroup();

rbDineIn = **new** RadioButton("Dine In");

rbDineIn.setSelected(**true**);

rbDineIn.setToggleGroup(tGrpDiningChoice);

rbTakeOut =...

RadioButton rad = (RadioButton)tGrpDiningChoice.getSelectedToggle();

String choice = rad.getText();

1. ComboBox – getItems.add(), getItems().addAll, setValue

ComboBox<String> cmb = new ComboBox<>() // Could be Integer, Double

cmb.getItems().add(str)

cmb.getItems().addAll(str1, str2, ...)

cmb.setValue(str)

String str = (String)cmb.getValue() // would use in event handler

1. ListView – getItems.add(), getItems().addAll, getSelectionModel().setSelectionMode,

**protected** ListView<String> lvwInterests = **new** ListView<>();

lvwInterests.getSelectionModel().setSelectionMode(SelectionMode.***MULTIPLE***);

lvwInterests.getItems().addAll("Music", "Dance", "Disc Golf", "Theater", "Reading");

**for**(String interest : lvwInterests.getSelectionModel().getSelectedItems()) {

interests += interest + ", ";

}

1. The *Slider* control is considered briefly in an [Appendix](#Appendix_Slider_Example), and the *Chart* is considered briefly in another [Appendix](#Appendix_Charts).

## Pane Summary

1. GridPane – add

GridPane gPane = new GridPane();

gPane.add(control/pane, column, row);

1. HBox, VBox – getChildren().add, getChildren.addAll

HBox hBox = new HBox();

hBox.getChildren().add(ctrl/pane);

hBox.getChildren().addAll(ctrl1, ctrl2);

VBox vBox = new VBox();

vBox.getChildren().add(ctrl/pane);

vBox.getChildren().addAll(ctrl1, ctrl2);

1. The *BorderPane* control is considered briefly in an [Appendix](#Appendix_BorderPane).

## Event Handler Summary

control.setOnAction( new MyEventHandler() )

public void handle(ActionEvent e) {

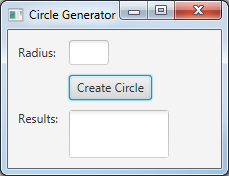
// 1. Get info supplied by user

// 2. Compose a message, calculate, etc

// 3. Update gui

}

# Exercises

1. Consider the GUI shown on the right.
2. Write a few lines of code to: (a) create a *GridPane*, (b) create the *TextArea*, (c) and place the *TextArea* in the position shown in the GUI on the right. Note: you are not creating the entire GUI, just the piece described.
3. When the button is pressed a Circle object is created using the radius from the text box. Then, its area is displayed in the *TextArea*. Assume: (a) all controls are accessible and (b) you have a *Circle* class with a constructor that takes a radius and a *getArea* method that returns the area of the Circle, (c) the name of the text box is *txtRadius*. Complete the *handle* method below to accomplish this.

private class CreateCircleEventHandler implements EventHandler<ActionEvent> {

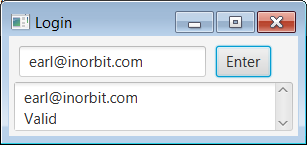
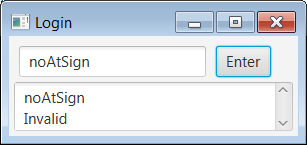
public void handle(ActionEvent event) {

// Write code here...

}

}

1. Write a line or two of code to register the inner class event handler from part *b* with the button. Assume the button’s name is *btnCreate.*
2. Consider the Gui shown below, on the right. The user types in an email address. When the button is pressed, if the email address has a ‘@’ character anywhere in the email address, then the email address is displayed along with “Valid” on the next line. Otherwise, if the email address does not contain a ‘@’ character, then the invalid email address and “Invalid” are displayed as shown below, on the left.

The partially written code for a Gui is shown on the next page. You will write the missing code in the two boxes so that the Gui functions as described. **Note: You must use the two panes that are defined in the code.**

**public** **class** Main **extends** Application {

TextArea txa = **new** TextArea();

TextField txf = **new** TextField("type email");

Button btn = **new** Button("Enter");

**public** **void** start(Stage primaryStage) {

HBox h = **new** HBox(); **// You must use this**

GridPane g = **new** GridPane(); **// You must use this**

|  |
| --- |
|  |

Scene scene = **new** Scene(g,225,90);

primaryStage.setScene(scene);

primaryStage.setTitle("Stringinator");

primaryStage.show();

}

**public** **static** **void** main(String[] args) {

*launch*(args);

}

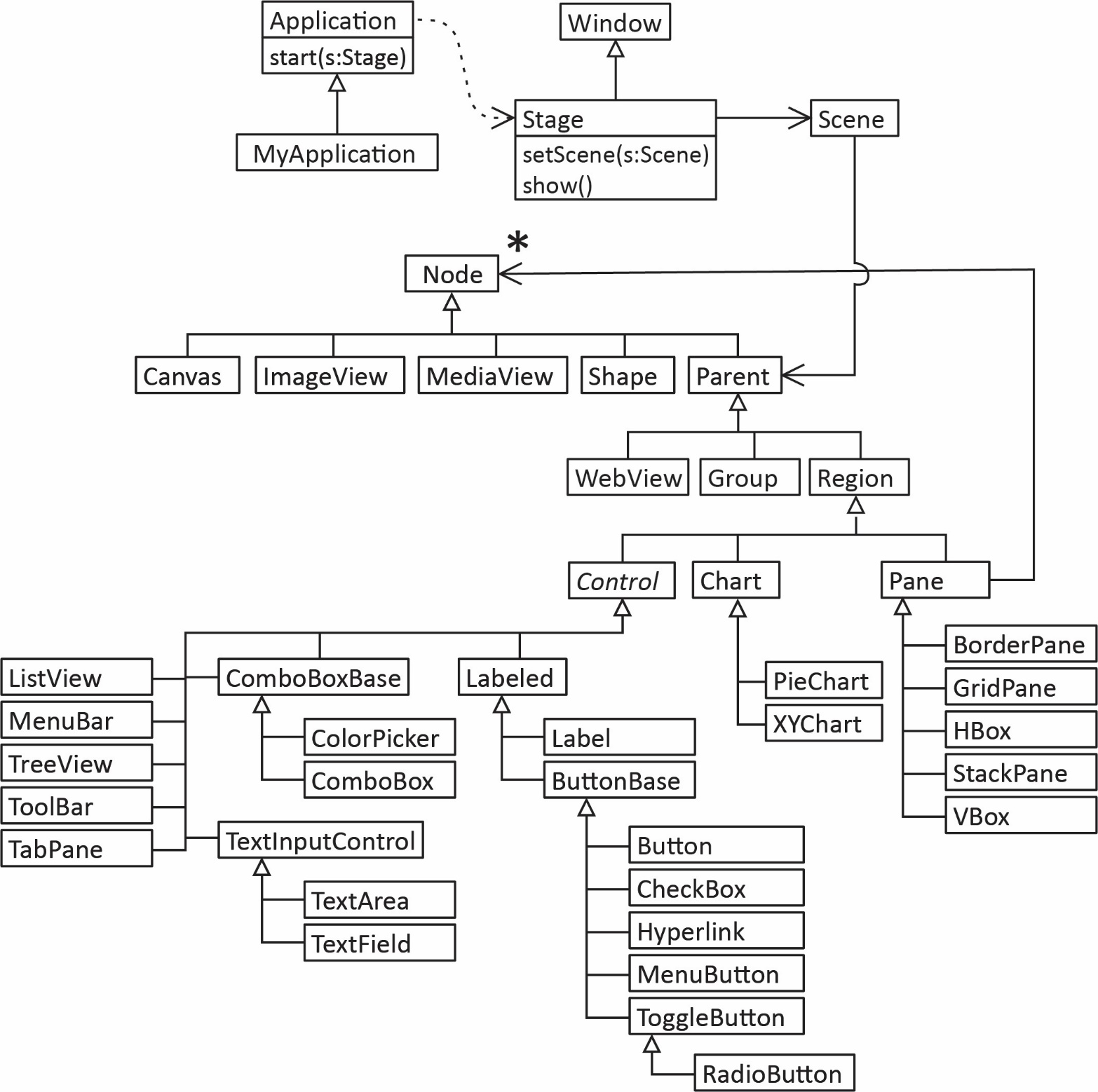
**private** **class** EH **implements** EventHandler<ActionEvent> {

**public** **void** handle(ActionEvent e) {

|  |
| --- |
|  |

Appendix

1. The Scene Class Hierarchy
2. The figure below shows the framework for building a JavaFX GUI. Note that a *Scene* can contain a *Pane*, but in general, the association is more general; the *Scene* has an association with the *Parent* class of which *Pane* is a subclass. Note, also that a *Pane* can have many *Node* instances where each *Node* instance can be a GUI Control, another *Pane*, or other items such as a *MediaView* object.



1. CSS in JavaFX
   1. Introduction

*Cascading Style Sheets* (*CSS*) is a technique for separating the styling (*look and feel*) of a GUI from its layout. We write Java code to build a GUI. We can write Java code, alongside the code to build the GUI, to style the GUI as well. However, it is considerably easier to do but construction and styling if we separate the two as much as possible. This is the basic idea of what CSS provides.

A *style sheet* is a text file usually saved with a *.css* extension. It contains *style definitions* which are named collections of *style rules.* Below, we show a style sheet. The *.label* style makes all *Label*s on our GUI blue. We explain more as we go along.

/\* style definition \*/

.root {

/\* style rules \*/

-fx-background-color:**palegoldenrod**;

-fx-padding: 10, 10, 10, 10;

-fx-font-size: 18pt ;

}

/\* style definition \*/

.label {

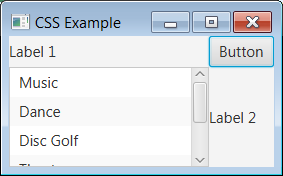
/\* style rules \*/

-fx-text-fill:blue;

}

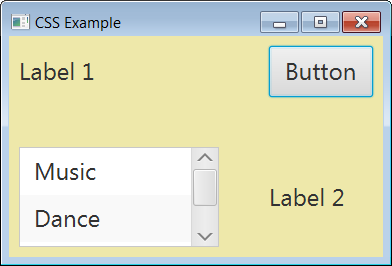
To apply a style sheet (*application.css*), we add a line of code like this in *start*.

scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());

JavaFX defines a [default style sheet](https://pastebin.com/0PebD9nR) (*caspian.css*). The GUI on the right does not define its own style sheet and so it is using the default style sheet. As we can see, things are a bit scrunched up together. To do any kind of modification to the style, we must create our own style sheet (or use code-not preferred).

* 1. Class Styles

A *class style* applies its style rules for all nodes of that class. The *.root* style definition below is a *class style*. *root* applies its style rules for all nodes in the root pane, including the root pane itself.

Consider the GUI on the right and the style sheet below.

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example1.java* and the corresponding style sheet is: *application1.css*. |

.root {

-fx-background-color:palegoldenrod;

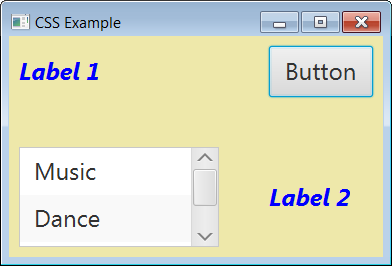
-fx-padding: 10, 10, 10, 10;

-fx-font-size: 18pt;

-fx-hgap: 50;

-fx-vgap: 50;

}

Consider the GUI on the right and the style sheet below.

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example2.java* and the corresponding style sheet is: *application2.css*. |

.root {

-fx-background-color:palegoldenrod;

-fx-padding: 10, 10, 10, 10;

-fx-font-size: 18pt;

-fx-hgap: 50;

-fx-vgap: 50;

}

.label {

-fx-text-fill:blue;

-fx-font-style: italic;

-fx-font-weight: bold;

}

Notes:

* *.label* is a *class style* and applies its style rules to all instances of *Label* in the GUI.
* Reference for colors: <https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html#typecolor>
* Reference for *.fx-font*: <https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html#fontprops>
* Label properties: <https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#label>

Class style definitions are named with the name of the JavaFX class in lower case. For example, you could define a style definition for all *Button* instances on your GUI with:

.button {

}

If the name of the class is compound, then you use dashes between the words. For example, to write a class selector for all *CheckBox* instances, you would write:

.check-box {

}

Class style definitions for *Pane* objects can be defined this way:

|  |  |  |
| --- | --- | --- |
| HBox {  } | VBox {  } | GridPane {  } |

The name of a style definition is also called a *selector* because it defines which nodes (controls) are *selected* to be styled by the definition.

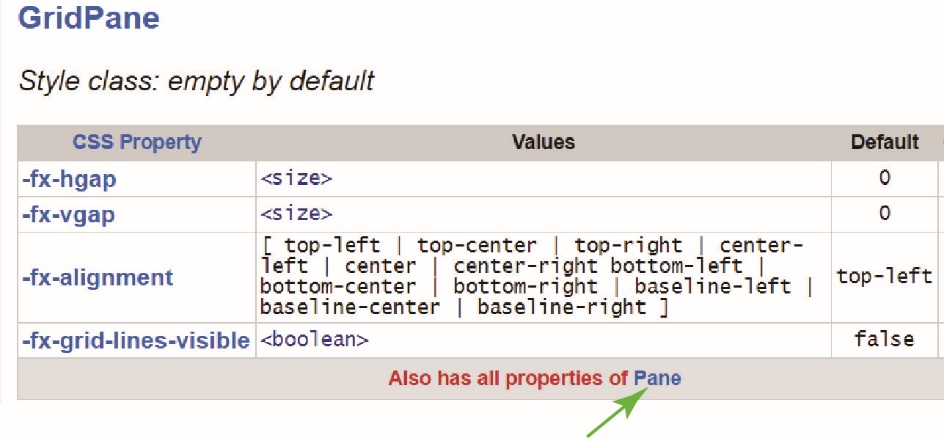
A style rule has this syntax:

css property: value;

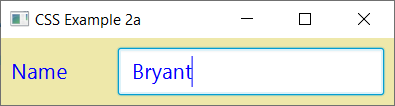
When you type the “:” in Eclipse, usually (but not always) a popup dialog will appear to help you choose a value.

A reference for CSS properties and allowable values is here:

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html>

For instance, to see what properties you can use for a *GridPane,* select the link for *GridPane* and the result is shown below. There, you see that there are only four properties. However, since *GridPane* is a subclass of *Pane,* you can use any of the ones defined for *Pane* and notice there is a link for *Pane* at the bottom.

You can also style different nodes with common properties

.label, .text-field {

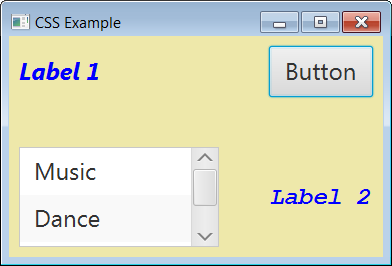
-fx-text-fill:blue;

-fx-font-size: 16pt ;

}

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example2a.java* and the corresponding style sheet is: *application2a.css*. |

* 1. Anonymous Selectors

An *anonymous* (custom) selector is one that can be applied to any node, and must be applied with code. For this type, you create a name for the definition. For example:

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example3.java* and the corresponding style sheet is: *application3.css*. |

.monosp {

-fx-font-family: "monospace";

}

Then, we apply it directly to the label (see figure on the right):

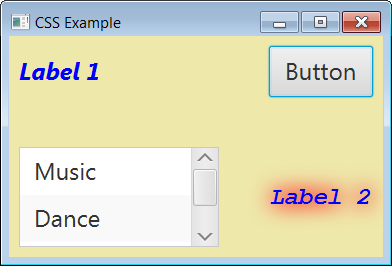
Label lbl2 = **new** Label("Label 2");

lbl2.getStyleClass().add("monosp");

We can apply this selector to any other nodes we want to.

You used a custom selector in the Lab on GUI construction, *h\_or\_v\_box* that we applied to all *HBox* and *VBox* instances which is shown below on the left. The style sheet on the right is equivalent, and has the benefit that all of the styles are class styles, thus, automatically applied. Either approach is enough for any GUI you do make for this class.

|  |  |
| --- | --- |
| .root {  -fx-padding: 5px;  }  .h\_or\_v\_box {  -fx-padding: 5px;  -fx-spacing: 5px;  }  .gridpane {  -fx-hgap:10px;  -fx-vgap:10px; | .root {  -fx-padding: 10px;  }  **GridPane** {  -fx-hgap:10px;  -fx-vgap:10px;  }  **HBox** {  -fx-spacing: 10px;  }  **VBox** {  -fx-spacing: 10px;  } |

Multiple custom selectors can be applied to the same node. For example:

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example4.java* and the corresponding style sheet is: *application4.css*. |

.shadow {

-fx-effect: **dropshadow(**three-pass-box**,** red**,** 30**,** 0.5**,** 0**,** 0**)**;

}

Now, we apply it to the label (along with *monosp*):

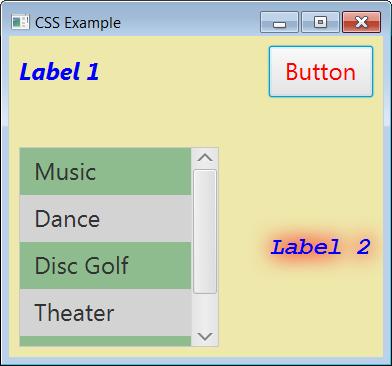
Label lbl2 = **new** Label("Label 2");

lbl2.getStyleClass().add("monosp");

lbl2.getStyleClass().add("shadow");

The result is shown on the right above.

* 1. Pseudo-class Selectors

A *pseudo-class* selector is one that applies to a particular state the node may be in. The name of the style definition has an added keyword to denote this state. For example, in a *ListView* we can make every other row a different color. Also, we can make the button change color when it is hovered over:

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example5.java* and the corresponding style sheet is: *application5.css*. |

.list-cell:even {

-fx-background-color:darkseagreen;

}

.list-cell:odd {

-fx-background-color:lightgray;

}

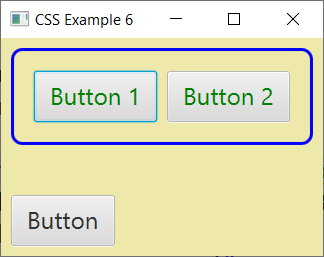
.button:hover {

-fx-text-fill: red;

}

Note that since *ListCell* is class in JavaFX representing a cell in the *ListView*, it is applied automatically.

* 1. Descendant Selectors

A *descendant* selector selects all nodes that are descendants of a specified selector. For example:

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example6.java* and the corresponding style sheet is: *application6.css*. |

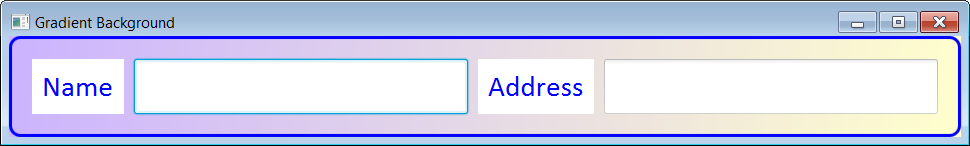
**HBox** > .button {

-fx-text-fill: green;

}

* 1. ID Selectors

An *ID* selector is a way to apply a style to exactly one element. This is done by using a “#” in front of the style name (which is the ID), and then setting the ID for a particular node. For example:



#hBoxStyle {

-fx-spacing:10.0;

-fx-background-color: **linear-gradient(**from 0.0% 0.0% to 100.0% 100.0%**,**

**rgb(**204**,**179**,**255**)** 0.0**,** **rgb(**255**,**255**,**204**)** 90.0**,** **rgb(**255**,**255**,**204**)** 100.0**)**;

-fx-fill: blue;

-fx-padding:20 20 20 20;

-fx-border-color: blue;

-fx-border-width: 3;

-fx-border-style:solid;

-fx-border-radius:10;

}

It is applied to the *HBox* which serves as the root pane in this way:

HBox hBox = **new** HBox();

hBox.setId("hBoxStyle");

|  |
| --- |
| This example is found in the code download for this chapter: *examples\_css/Example7.java* and the corresponding style sheet is: *application7.css*. |

* 1. Practical Issues

Working with CSS can be tedious and confusing. Sometimes you will write a rule and it doesn’t appear to do anything. This can mean a number of things:

* The property is not valid for the selector. In this case, you don’t get an error, it just doesn’t do what you are expecting. For example, in the example above for the *label* selector, we defined the text color as blue:

-fx-text-fill:blue;

However, if you remove it from *label* and put it in *root* the text will not appear blue. (Strangely, that is not a valid property for *root*.)

* Sometimes, as you make changes to a CSS file, and save it each time, the rule doesn’t work. But, if you close the GUI, and run again, it will work. It is like the GUI is using a cached version of the CSS file. I have been very frustrated with this behavior and have even resorted to closing Eclipse and restarting which did work.
* Sometimes the rules you define with different selectors conflict. In this case, it is useful to comment out a bunch of rules to isolate what is causing the problem. Use: /\* \*/ to surround what you want to comment out.

–

* 1. Motivation for CSS

You can think of a GUI as having four elements:

1. Structure – the layout of the GUI and the controls.
2. Content – the data/information that is in controls (*e.g.* names in a *ComboBox*) or in memory.
3. Style – the *look and feel* of a GUI (*e.g.* borders, colors, fonts, *etc.*).
4. Event Handlers – the code to run when events are triggered by the user.

All these elements have a distinctly different *purpose* but are interdependent*.* In computer science, we like to put elements with differing purposes, when possible, into their own module (class, file, method, package, *etc.*).

If we provide style in our GUI by using code (*e.g.* myButton.setPadding(…)) then we are coupling *style* and *structure.* Think about it, for each control, you could easily have 5 or more lines of code to provide styling – that would be hard to read, hard to maintain. And think about all the duplication, *e.g.* set the padding on all elements to the same value. To reuse the GUI and change the styling would be very tedious.

In the remainder of this section we show how to separate style from structure by utilizing CSS (cascading style sheets). The basic idea is to define style *rules* in a file separate from the source code, and then use code to link the rules to controls and panes.

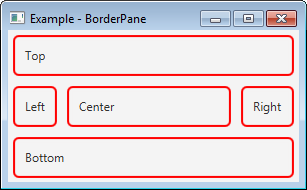
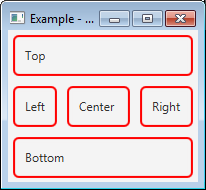
A good read on JavaFx CSS is:

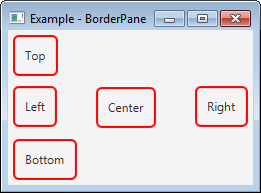
<https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/apply-css.htm>

1. BorderPane

The example in this section is in the *examples\_panes* package, *BorderPaneExample* class.

[*BorderPane*](https://docs.oracle.com/javase/9/docs/api/javafx/scene/layout/BorderPane.html)is used as the default root pane when you create a JavaFX project. I have not found much use for it and typically replace it, probably because I don’t understand how it sizes elements. However, you will see it in examples on the web so we will cover it briefly. *BorderPane* defines five regions (top, right, bottom, left, center) in which to store a node. The GUI below on the left uses a *BorderPane* to hold five *Labels.* On the right, is the same GUI, but the window is stretched horizontally showing that some of the nodes resize automatically.



The essence of the code for this example is shown below. However, without some styling, it will look as shown on the right.

BorderPane brdRootPane = **new** BorderPane();

Label lblTop = **new** Label("Top");

Label lblRight = **new** Label("Right");

Label lblBottom = **new** Label("Bottom");

Label lblLeft = **new** Label("Left");

Label lblCenter = **new** Label("Center");

brdRootPane.setTop(lblTop);

brdRootPane.setRight(lblRight);

brdRootPane.setBottom(lblBottom);

brdRootPane.setLeft(lblLeft);

brdRootPane.setCenter(lblCenter);

The styling I applied is shown below. I’m sure you can do this with CSS, but I developed this example before I leaned CSS in JavaFX:

lblTop.setMaxWidth(Double.***MAX\_VALUE***);

lblRight.setMaxWidth(Double.***MAX\_VALUE***);

lblBottom.setMaxWidth(Double.***MAX\_VALUE***);

lblLeft.setMaxWidth(Double.***MAX\_VALUE***);

lblCenter.setMaxWidth(Double.***MAX\_VALUE***);

1. Slider Example

The example in this section is in the *examples\_controls* package, *SliderExample* class.

The GUI on the right utilizes a [*Slider*](https://docs.oracle.com/javase/9/docs/api/javafx/scene/control/Slider.html)control. It is constructed with this code:

sldSatisfaction = **new** Slider();

sldSatisfaction.setMinWidth(400);

sldSatisfaction.setMin(-5.0);

sldSatisfaction.setMax(5.0);

sldSatisfaction.setValue(0);

sldSatisfaction.setShowTickLabels(**true**);

sldSatisfaction.setShowTickMarks(**true**);

sldSatisfaction.setMajorTickUnit(1);

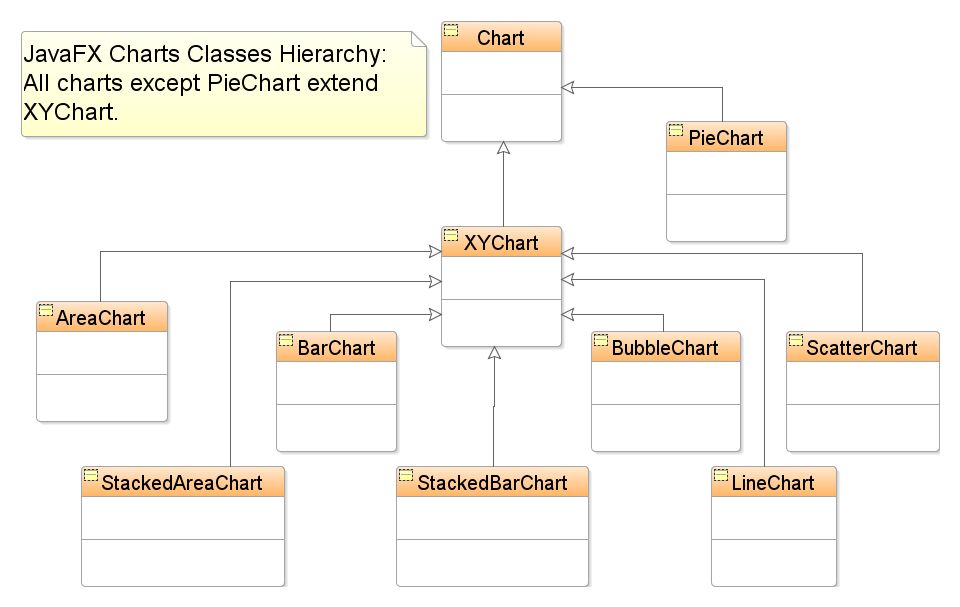
sldSatisfaction.setMinorTickCount(1);

sldSatisfaction.setSnapToTicks(**true**);

The *Slider’s getValue* method is used in an event-handler to obtain the value selected.

1. Charts

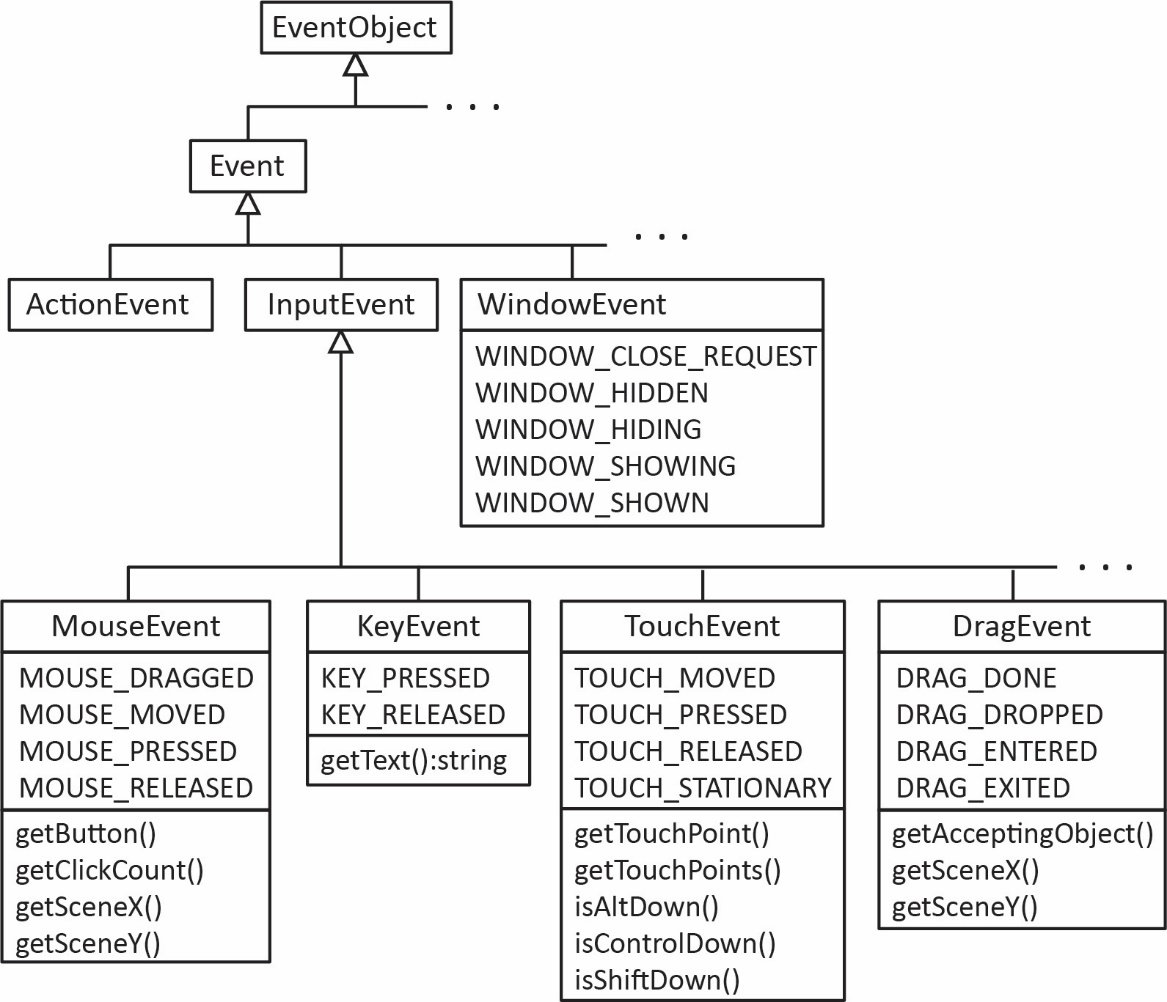
The class diagram below shows the *Chart* hierarchy. You can think of a *Chart* as a subclass of *Control*; however, technically, *Chart* and *Control* are subclasses of *Node*. The code download that accompanies these notes shows examples of creating various charts.



Source: <https://www.javacodegeeks.com/wp-content/uploads/2013/01/JavaFXChartClassDiagram.png>

1. JavaFX Event Handling

The (partial) *Event* hierarchy is shown below. The only event that we consider is *ActionEvent* and we won’t directly use the event object itself that is passed to the event-handler. Other types of events provide a rich set of properties and methods to provide more information about the event. For example, a *MouseEvent* reveals whether the mouse was dragged, moved, a button was pressed, a button was released, which button was pressed, how many times the button was clicked, the location of the mouse relative to the scene, *etc.*



The *EventHandler* interface is generic and can be implemented with many different types of events:

**interface** EventHandler<T **extends** Event> {

**void** handle(T event);

}

The abstract *Node* class, which is a superclass for the controls we considered, provides a number of methods for hooking into various events that can occur as shown below. The parameters for these methods are generic and can be implemented with many different types of events

setOnContextMenuRequested, setOnDragDetected, setOnDragDone, setOnDragDropped, setOnDragEntered, setOnDragExited, setOnDragOver, setOnInputMethodTextChanged, setOnKeyPressed, setOnKeyReleased, setOnKeyTyped, setOnMouseClicked, setOnMouseDragEntered, setOnMouseDragExited, setOnMouseDragged, setOnMouseDragOver, setOnMouseDragReleased, setOnMouseEntered, setOnMouseExited, setOnMouseMoved, setOnMousePressed, setOnMouseReleased, setOnRotate, setOnRotationFinished, setOnRotationStarted, setOnScroll, setOnScrollFinished, setOnScrollStarted, setOnSwipeDown, setOnSwipeLeft, setOnSwipeRight, setOnSwipeUp, setOnTouchMoved, setOnTouchPressed, setOnTouchReleased, setOnTouchStationary, setOnZoom, setOnZoomFinished, setOnZoomStarted

1. Inner Classes
2. **Inner Class** – An *inner class* is a class that is a *member* of another class. In other words, it is a class defined inside another class.

**public** **class** Foo {

**public** Foo() {

InnerFoo innerFoo = **new** InnerFoo();

}

...

**public** **class** InnerFoo {

**public** InnerFoo() {...}

...

}

}

1. **Advantages** – In some situations an inner class can make a program simpler for two reasons:

* An inner class can reference the all members of the outer class (including private).
* When the inner class is only needed inside the outer class. In other words, other classes in the system do not need to reference the inner class. In this case we would make the inner class *private*. However, other classes *can* access the inner class if needed, as long as it is *public*. Thus it provides better encapsulation.

1. **Features of Inner Classes**:

* An inner class can be declared public, protected, or private subject to the same visibility rules applied to a member of the class.
* An inner class is compiled into a class named: *OuterClassName*$*InnerClassName.class*. For the example above: *Foo$InnerFoo.class*
* An inner class can be declared static. A static inner class can be accessed using the outer class name. A static inner class cannot access non-static members of the outer class.
* Objects of an inner class are often created from within the outer class (as in the example above). However, you can also create an object of an inner class from another class:

Foo foo = new Foo();

Foo.InnerFoo innerFoo = foo.new InnerFoo();

1. Event Handling using Anonymous Inner Classes
2. **Anonymous Inner Class** – An anonymous inner class is a class:

* Is a class without a name
* Is defined inside another class
* Extends a superclass and/or implements an interface
* Must immediately be instantiated as it is defined

1. **Example** – An anonymous inner class event handler for the hello world application considered previously.*:*

btnHelloWorld.setOnAction(

// Create an instance of...

**new**

// Anonymous inner class

EventHandler<ActionEvent>() {

**public** **void** handle(ActionEvent e) {

String name = txfName.getText();

txaMessage.setText(name + ", Hello World!");

}

}

);

Which we usually write more compactly:

btnHelloWorld.setOnAction( **new** EventHandler<ActionEvent>() {

**public** **void** handle(ActionEvent e) {

String name = txfName.getText();

txaMessage.setText(name + ", Hello World!");}});

1. **More on Anonymous Inner Classes:**

* Can access instance variables of the outer class as well as local variables defined by the enclosing method.
* Compiled into a class named OuterClassName$*n*.class. In the example above, the main class is *Ex1b\_HelloWorld.class* and the anonymous inner class is *Ex1b\_HelloWorld$1.class*. If there were more anonymous inner classes they would be numbered 2, 3, *etc*.

1. Event Handling using Lambda Expressions
2. **Example** – Event handling can also be implemented with a *lambda expression*, which is a new feature of Java 8*.* A *lambda expression lets* you create an anonymous instance of a single-method class more compactly*.*

btnHelloWorld.setOnAction(e -> {

String name = txfName.getText();

txaMessage.setText(name + ", Hello World!");

});

Or, we could write a method:

**public** **void** helloWorldEventHandler() {

String name = txfName.getText();

txaMessage.setText(name + ",asf Hello World!");

}

And then use a lambda expression to call the method:

btnHelloWorld.setOnAction(e -> helloWorldEventHandler() );

1. **Explanation** –

“The compiler treats a lambda expression as if it is an object created from an anonymous inner class. In this case, the compiler understands that the object must be an instance of *EventHandler<ActionEvent>*. Since the *EventHandler* interface defines the *handle* method with a parameter of the *ActionEvent* type, the compiler automatically recognizes that *e* is a parameter of the *ActionEvent* type, and the statements are for the body of the *handle* method. The *EventHandler* interface contains just one method. The statements in the lambda expression are all for that method. If it contains multiple methods, the compiler will not be able to compile the lambda expression. So, for the compiler to understand lambda expressions, the interface must contain exactly one abstract method. Such an interface is known as a *functional interface* or a *Single Abstract Method (SAM) interface*.”[[3]](#footnote-3)

1. **Other Uses** – You can do many other things with Lambda Expressions:

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

Here are some examples that I will explain in class:

emps.forEach(e -> e.setAge(e.getAge()+1));

emps.removeIf(e -> e.getAge()>45 && e.getAge()<60);

**int** sum = emps.stream()

.mapToInt(Employee::getAge)

.sum();

List<Employee> emps2 = emps.stream()

.filter(e -> e.getAge()<40)

.collect(Collectors.*toList*());

emps.sort(Comparator.*comparingInt*(Employee::getAge)

.thenComparingDouble(Employee::getSalary));

Collections.*sort*(emps, (e1,e2) -> e1.getAge()-e2.getAge() );

printPersons(

roster,

// Lambda expression

(Person p) -> p.getGender() == Person.Sex.MALE

&& p.getAge() >= 18

&& p.getAge() <= 25

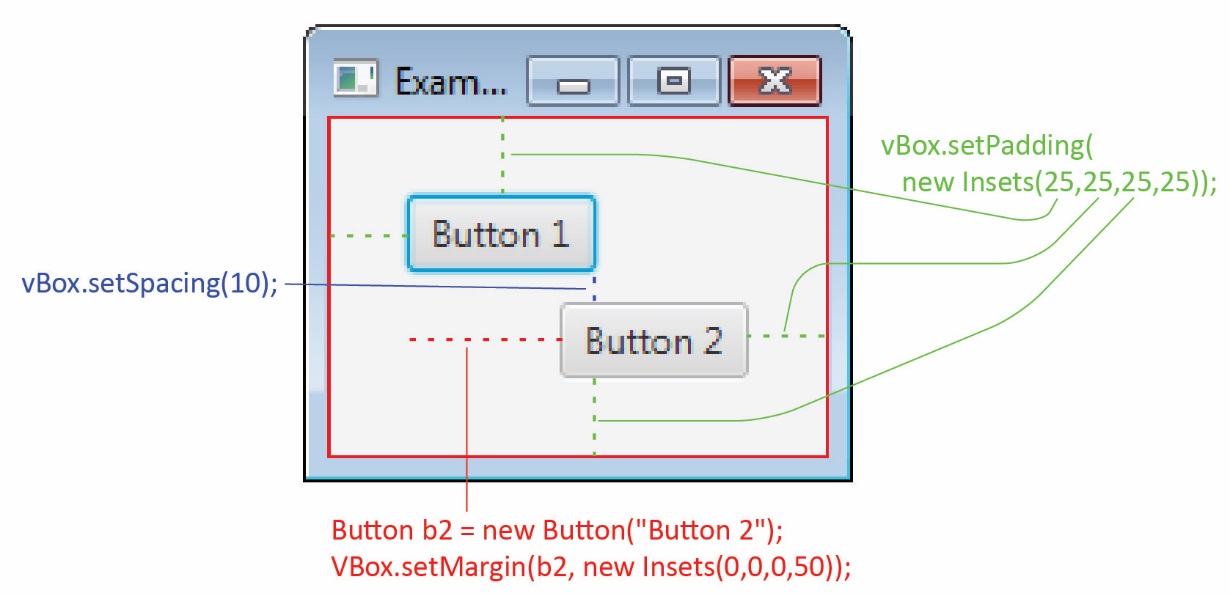
);

1. Styling A GUI with Code

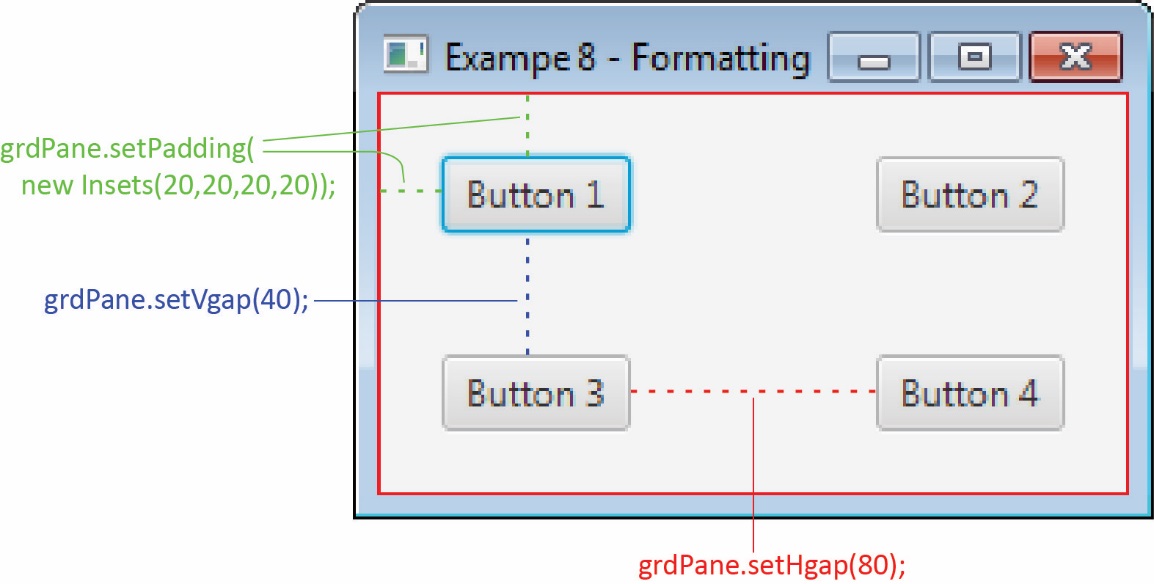
Note: most (if not all) of the things discussed here can be done with CSS. These notes were developed before I taught CSS. However, the methods used to set properties all have CSS properties that are the same. It is best practice to use CSS when possible.

1. **Example** – In the examples considered so far, the actual code shown in these notes has not used any formatting (to make things simpler); however, the screen shots and the code downloads do have formatting. Below are a few methods for setting:

* **Padding –** All *Pane* objects have a *setPadding* method which accepts an *Inset* object which defines how far controls are inset from the border of the pane. For example, you can set the padding on a *Vbox* as shown in the example below.
* **Spacing –** *VBox* and *HBox* objects have a *setSpacing* method defines the space between controls as shown in the figure below. The other *Pane* objects we study do not have this method.
* **Margin –** All *Pane* objects have a static *setMargin* method which accepts the control to set the margin for and an *Inset* object which defines the additional space between the padding or spacing. The example below shows that the left margin has been set for the button, *b2*.



1. **Example** – The *GridPane* has a *setHGap* and *setVGap* methods for setting the horizontal and vertical spacing between controls as shown in the example below:

****

1. **Example** – If the controls are different sizes they will render by default as shown on the left below. On the right we make controls the same width for each column:

|  |  |
| --- | --- |
| **E:\Data-Classes\CS 1302 - Programming 2-Spring 17\notes\07_ch14_16_GUI\b4.jpg** | **E:\Data-Classes\CS 1302 - Programming 2-Spring 17\notes\07_ch14_16_GUI\b5.jpg**  Button b1 = **new** Button("Add");  b1.setMaxWidth(Double.***MAX\_VALUE***);  Button b2 = **new** Button("Delete");  b2.setMaxWidth(Double.***MAX\_VALUE***);  Button b3 = **new** Button("Generate Report");  b3.setMaxWidth(Double.***MAX\_VALUE***);  Button b4 = **new** Button("Exit");  b4.setMaxWidth(Double.***MAX\_VALUE***); |

1. **Example** – To make all the controls the same size takes a bit more work.

|  |  |
| --- | --- |
| E:\Data-Classes\CS 1302 - Programming 2-Spring 17\notes\07_ch14_16_GUI\b6.jpg | // Determine width of largest item  Group root = **new** Group();  Scene scene = **new** Scene(root);  Button b3\_dummy = **new** Button("Generate Report");  root.getChildren().addAll(b3\_dummy);  root.applyCss();  root.layout();  **double** maxWidth = b3\_dummy.getWidth();  System.***out***.println(maxWidth);  // Build GUI  Button b1 = **new** Button("Add");  Button b2 = **new** Button("Delete");  Button b3 = **new** Button("Generate Report");  Button b4 = **new** Button("Exit");  b1.setPrefWidth(maxWidth);  b2.setPrefWidth(maxWidth);  b3.setPrefWidth(maxWidth);  b4.setPrefWidth(maxWidth);  grdPane.add(b1, 0, 0);  grdPane.add(b2, 1, 0);  grdPane.add(b3, 0, 1);  grdPane.add(b4, 1, 1); |

1. **Example** – Right alignment in a *VBox*. Also illustrates the red border that was in all the examples above.

|  |  |
| --- | --- |
| E:\Data-Classes\CS 1302 - Programming 2-Spring 17\notes\07_ch14_16_GUI\b7.jpg | VBox vBox = **new** VBox();  vBox.setPadding(**new** Insets(20, 20, 20, 20));  vBox.setSpacing(10);  vBox.setAlignment(Pos.***CENTER\_RIGHT***);  vBox.setBorder(**new** Border(**new** BorderStroke(  Color.***RED***, BorderStrokeStyle.***SOLID***, **null**, **null**)) );  // Build GUI  Button b1 = **new** Button("Add");  Button b2 = **new** Button("Delete");  Button b3 = **new** Button("Generate Report");  Button b4 = **new** Button("Exit");  vBox.getChildren().addAll(b1,b2,b3,b4); |

1. **Example** – Two *VBox’s* in an *HBox.*

|  |  |
| --- | --- |
| E:\Data-Classes\CS 1302 - Programming 2-Spring 17\notes\07_ch14_16_GUI\b8.jpg | // Left VBox  VBox vBoxL = **new** VBox();  vBoxL.setPadding(**new** Insets(20, 20, 20, 20));  vBoxL.setSpacing(10);  vBoxL.setAlignment(Pos.***CENTER\_RIGHT***);  vBoxL.setBorder(**new** Border(**new** BorderStroke(Color.***RED***, BorderStrokeStyle.***SOLID***, **null**, **null**)) );  Button b1 = **new** Button("Add");  Button b2 = **new** Button("Delete");  vBoxL.getChildren().addAll(b1,b2);  // Right VBox  VBox vBoxR = **new** VBox();  vBoxR.setPadding(**new** Insets(20, 20, 20, 20));  vBoxR.setSpacing(10);  vBoxR.setAlignment(Pos.***CENTER\_RIGHT***);  vBoxR.setBorder(**new** Border(**new** BorderStroke(Color.***RED***, BorderStrokeStyle.***SOLID***, **null**, **null**)) );  Button b3 = **new** Button("Generate Report");  Button b4 = **new** Button("Exit");  vBoxR.getChildren().addAll(b3,b4);  // Root container, HBox  HBox hBox = **new** HBox();  hBox.setPadding(**new** Insets(20, 20, 20, 20));  hBox.setSpacing(10);  hBox.setBorder(**new** Border(**new** BorderStroke(Color.***BLUEVIOLET***, BorderStrokeStyle.***SOLID***, **null**, BorderStroke.***THICK***)) );  hBox.getChildren().addAll(vBoxL, vBoxR); |

1. Sizing elements, and *how* different panes layout controls is rather involved. If you are interested, here is a reference:

<http://docs.oracle.com/javafx/2/layout/size_align.htm>

1. Resources

|  |  |
| --- | --- |
| **Source** | **Description** |
| [JavaFX Tutorials](https://docs.oracle.com/javase/8/javase-clienttechnologies.htm) | Oracle tutorials |
| [Implementing JavaFX Best Practices](https://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.htm) | Custom preloader, meaningful package names, MVC with FXML, CSS, Run Tasks on a Background Thread |
| [Best Practices for Efficient Development of JavaFX Applications](http://accelconf.web.cern.ch/AccelConf/icalepcs2017/papers/thapl02.pdf) | Conference paper. FXML, Scene Builder, MVC, GUI Testing |
| [JavaFX Tutorial](https://jenkov.com/tutorials/javafx/index.html) |  |

1. <https://en.wikipedia.org/wiki/User_interface_design> [↑](#footnote-ref-1)
2. See the [Anonymous inner classes](#Appendix_Anonymous_Inner_Classes) Appendix, and the [Lambda expressions](#Appendix_Lambda_Expressions) Appendix. [↑](#footnote-ref-2)
3. Intro to Java Programming, 10th ed., Liang, p.598 [↑](#footnote-ref-3)