JavaFX by Example

Introduction

In this programming text you will be introduced to the workings of JavaFX though the use of full code programming examples. These are intended to be small programs that you can run by yourself to see how the applications and therefore JavaFX functions as a GUI toolkit. All GUI toolkits use the same principles for handling events and drawing operations. Therefore if you can learn to use a single GUI toolkit you will easliy transfer that knowledge to other GUI toolkits without issue.

In all of these examples I will be using the Eclipse development environment and the Java 8 SDK. It is highly recommended that you do not copy paste these examples directly. If you copy paste these directly you will learn very little from the examples. My suggestion for best learning how these examples work is to write them out line by line yourself. You will start to notice common patterns and you will learn the toolkit quicker. It may take longer to get through the examples but in the long run you will save yourself a lot of time.

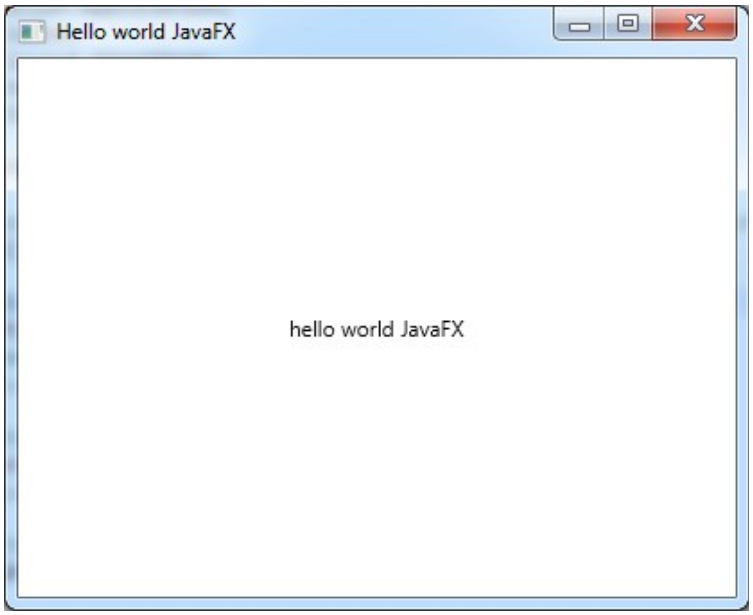
I will break everything down into steps and I will tell you the points at which an example should be fully compilable. You should take advantage of this as you will be able to see how the components we add will enable functionality on our test applications. It will also help you learn further.

JavaFX is going to replace the Java Swing and AWT libraries. At the moment JavaFX is external to the JDK but is included by default. Swing and AWT were around all the way back in 2002 (AWT is even older) so it is out of date and in dire need of replacement

So without further ado let us get into the examples

Example 001: Hello World with JavaFX

In this example you will build a simple hello world application. It will show you the minimum required structure to build a JavaFX application. You will see this structure over and over again in the examples thus it should be one of the first things you should start to learn and build. This example requires a lot of explaination as it will introduce a lot of concepts. Below is a screen shot of what we are building.



01) start a new Java Project and create a single source file called Example001.java and populate it with the following code

```
// imports necessary for this application to work
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class Example001 extends Application {

        // the init method that will initialise stuff before we start the running of our
        // application
        public void init() {}

        // the start method that will be the entry point for our application
        public void start(Stage primaryStage) {
```

```
                // code for 03

                // code for 04

        }

        // the stop method that will do clean up work when the application is finishing
        public void stop() {}

        // main method that will start our application
        public static void main(String[] args) {
                // code for 02
        }

}
```

Explaination

- What you see here is the basic shell of a JavaFX application. All JavaFX applications must extend the Application class. This will provide your application with all of the necessary support structures necessary for interacting with your user through the use of the keyboard and mouse. it will also provide the support structures that will enable your application to draw on the screen.
- The init() method is not required but is recommended. You can initialise almost everything before the application is run in this method. The only things you are not permitted to initialise here are Stage objects and Scene objects. These must be initialised the in the start() method
- The stop() method is called when your application is going to quit. This is a good place for you to free up resources (close files, network connections etc) before your application finishes. It is extremely bad practice to expect the OS to free all of your resources for you.
- Application is an abstract class that requires you implement the start method this is where you will construct your stages and scenes that will make up your GUI application. A stage is simply a top level window with decoration, a title, and window buttons (maximise, minimise, and close) Scene objects will be used to populate Stages with drawing and content. We call these Scenes because JavaFX uses a Scene Graph datastructure to represent graphical content (we will explore the scene graph in a lot more detail later)
    - The primaryStage represents the window that JavaFX will construct and provide you by default. it is possible to create more windows but we will do this at a later stage

02) replace the comment "code for 02" with this

```
launch(args);
```

Explaination

- If the application is launched this must be called in the Main method to tell JavaFX to start our application through the start() method

03) replace the comment "code for 03" with this

```
// set the name of our stage/window
primaryStage.setTitle("Hello world JavaFX");

// set the stage using a scene and put a basic layout on the scene
StackPane root = new StackPane();
primaryStage.setScene(new Scene(root, 400, 300));
```

Explaination

- Fairly simply first line. This sets the title of our window that you will see on the window decoration in the screenshot above
- The StackPane is an example of a layout. We will get onto layouts in a few examples time. We use the StackPane here purely for centring our text.
- The following line generates a Scene object and sets it on our stage. A scene takes three parameters. The first parameter specifies the root node of our scene graph. This will be used to render our window, The second and third parameters specify the width and height of the window in pixels.

04) replace the comment "code for 04" with this

```
// add some text to the pane
root.getChildren().add(new Label("hello world JavaFX"));

// show the stage
primaryStage.show();
```

Explaination

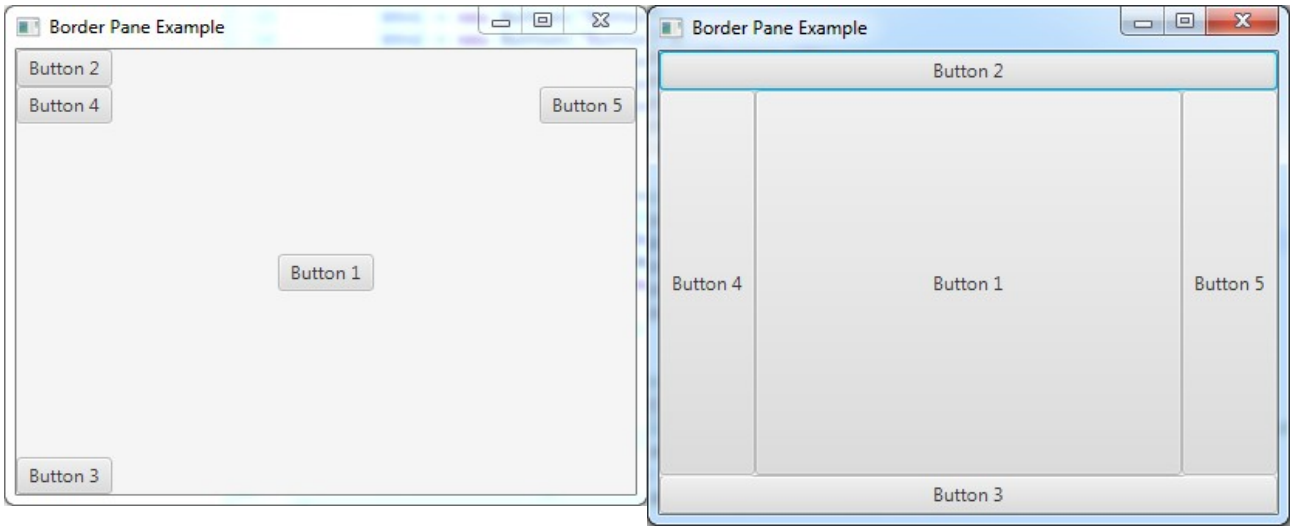
- here we are taking the root node of our scene graph (our stack pane) and adding a GUI element to display a piece of text into the scene graph
- The second line asks the primaryStage to make itself visible to the user

NOTES:

If you find that Eclipse is stating that it cannot find anything when you import anything from the JavaFX library this is because Java 8 integration in Eclipse is not fully production ready. There may be Eclipse updates during the semester that may fix this. In the meantime the easiest way around this problem is to go to your project properties and under java build path under libraries remove and readd your JRE System Library.

Example 002: Introducing the Button and the BorderPane layout

In this example you will be introduced to the Button class and also the first of the layouts we will use in our programming the BorderPane. We will create a BorderPane consisting of 5 buttons that will look like the screenshots below. The screenshot on the right is the result we are after but the screenshot on the left will be used to illustrate a point midway through development of this application.



01) start a new java project and create a single source file called Example002.java and give it the following shell code.

```
// simple example that introduces the border pane with the use of buttons

// imports
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

// class definition
public class Example002 extends Application {

	// the init method
	public void init() {
		// code 02

		// code 05
	}

	// the start method
	public void start(Stage primaryStage) {
		// code 03
```

```
                // code 04

        }

        // the stop method
        public void stop() {}

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

        // private fields of the class
        Button btn1, btn2, btn3, btn4, btn5;          // a few buttons that we are going to use for
the border pane
}
```

Explaination

- Here we have the basic shell of our application again. This is all pretty similar to the previous example except for one minor change. We have declared 5 buttons as fields of the class because we will use the init() method to initialise them and attach them to the scene in the start() method

02) replace the comment "code for 02" with the following.

```
// initialise all of our buttons
btn1 = new Button("Button 1");
btn2 = new Button("Button 2");
btn3 = new Button("Button 3");
btn4 = new Button("Button 4");
btn5 = new Button("Button 5");
```

Explaination

- This is the normal way a Button is constructed in JavaFX. There are other constructors available but this one will be used the most. It takes in a single String which will is the text to be displayed on the button.

03) replace the comment "code for 03" with the following

```
// set the tile on the stage, create a border pane and put it as the root
// of the scene
primaryStage.setTitle("Border Pane Example");
BorderPane bp = new BorderPane();
primaryStage.setScene(new Scene(bp, 400, 300));
```

Explaination

- like in the previous example the first things we will do with our primary stage is set the title, size and a root node for our scene graph. In this case we will use a BorderPane as the root of our scene graph.

04) replace the comment "code for 04" with the following code (code should be fully compilable here and I suggest you run it)

```
// add in the buttons to the border pane
bp.setCenter(btn1);
bp.setTop(btn2);
bp.setBottom(btn3);
bp.setLeft(btn4);
bp.setRight(btn5);

// show the stage
primaryStage.show();
```

Explaination

- Here we are adding the Buttons to our BorderPane before showing the stage. If you run this code now you will recreate the left screenshot at the start of this example.
- The BorderPane defines 5 areas: center, top, bottom, left, right. you may put whatever you want in these zones (could be widgets or other layouts, note how they match the method names too). You will notice however that there is a lot of free space around the buttons and their positioning does not make sense. This is because Buttons are configured by default to take the minimum amount of space to display their text.

05) replace the comment "code for 05" with the following (good time to run the code)

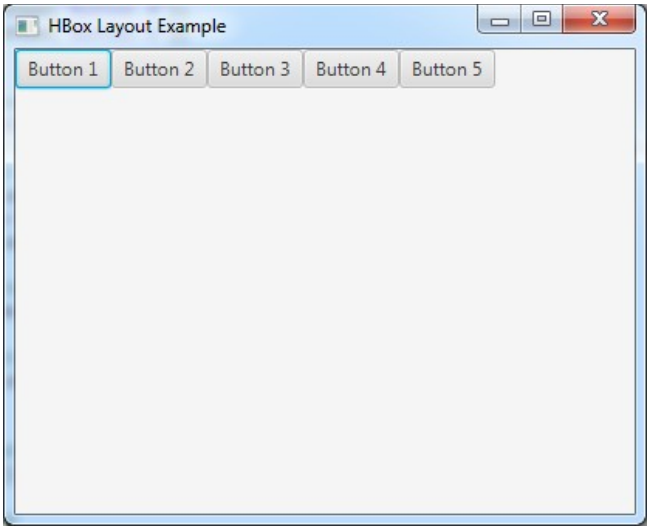
```
// set the maximum width and height of all buttons
btn1.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
btn2.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
btn3.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
btn4.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
btn5.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
```

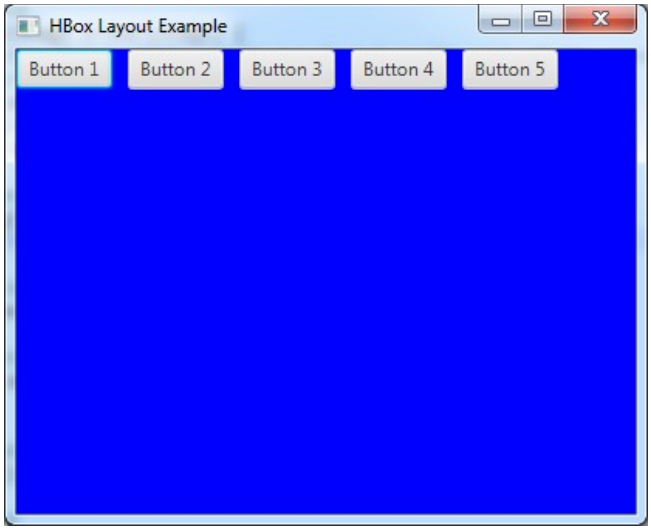
Explaination

- Here we are telling each button that the maximum size it should take (space permitting) is the maximum value that can be represented in a double (aprrox 1 x 10^308). The value is set once for the width and once for the height. This creates buttons that will expand into any available free space and will create the second screenshot.
- Try resizing the window after compilation. You will note that the center area will always expand in both directions to take available space. The top and bottom areas will take the minimum necessary height but will always expand to take the maximum horizontal space while the left and right areas will take the minimum necessary width but will expand to take the full available height. This is to ensure that the border is complete at all times.

Example 003: introducting the HBox layout

In this example we will introduce the Horizontal Box layout (HBox for short). this will arrange all items horizontally on the same line. This is one of the most common layout styles used in GUI programming, and thus can be found in almost every GUI toolkit in existance. In this example we will also see how to introduce spacing between widgets and also how show you how you can use CSS to style your widgets. The screenshots below show what a HBox looks like by default, and how it looks when some CSS styling and spacing is put on the HBox





01) start a new Java project and create a single source file called Example003.java and give it the following base code.

```
// imports
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

// class definition
public class Example003 extends Application {
	// the init method
	public void init() {
		// initialise the 5 buttons
		btn1 = new Button("Button 1");
		btn2 = new Button("Button 2");
		btn3 = new Button("Button 3");
		btn4 = new Button("Button 4");
		btn5 = new Button("Button 5");
	}

	// the start method
	public void start(Stage primaryStage) {
		// 02 code
	}
```

```
        // the stop method
        public void stop() {}

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

        // 5 private buttons to demonstrate the hbox layout
        private Button btn1, btn2, btn3, btn4, btn5;
}
```

Explanation:

- this is the same basic structure that you saw in the last example for generating and initialising 5 buttons.

02) replace the comment "02 code" with the following code (good point to compile and run code)

```
// set the title of our stage, create our layout as the root of the scene,
// and set our stage size
primaryStage.setTitle("HBox Layout Example");
HBox hb = new HBox();
primaryStage.setScene(new Scene(hb, 400, 300));

// 03 code

// add the 5 buttons to the hbox layout
hb.getChildren().addAll(btn1, btn2, btn3, btn4, btn5);

// show the stage
primaryStage.show();
```

Explanation:

- here we have our setup of the stage, except instead of setting a stack pane as the root of our scene graph we are setting a HBox as the root.
- note the hb.getChildren().addAll() method call. This method can take in any number of parameters and is a handy way of adding all of your controls in a single go. The order that items are added to the HBox will be shown in left to right order when the HBox is displayed.
- when you run this code you should get the left screenshot above

03) Replace the comment "03 code" with the following code (good place to run and compile code)

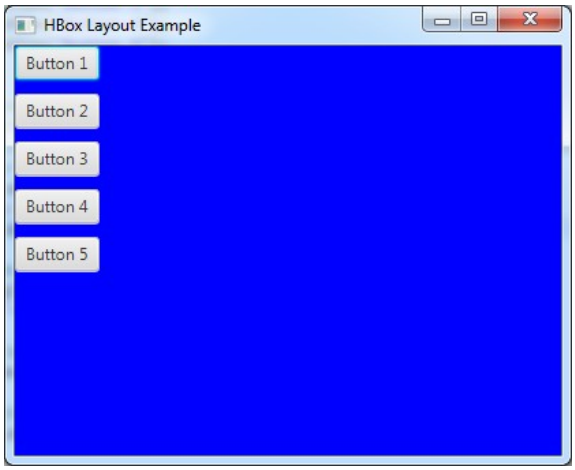
```
// add in some spacing to the hbox and set the background colour
hb.setSpacing(10);
hb.setStyle("-fx-background-color: #0000FF");
```

Explaination

- setSpacing() takes in a double value that represents the number of pixels that should appear between two components
- setStyle() permits you to add in CSS strings to style your application. In this example we are asking JavaFX to set the background colour of this HBox to be blue
- when you run the code here you should end up with the right screenshot above

Example 004: introducing the VBox layout

In the previous example we introduced the HBox layout. In this example we will introduce its cousin the VBox layout. You can probably guess that the VBox is the vertical box layout and by adapting the code produced in Example 003 you will be able to produce this screenshot below



01) Create a new java project with a single source file called Example004.java

02) take a complete copy of the Example 003 code and paste it into our source file.

03) modify the following lines from these

```
// set the title of our stage, create our layout as the root of the scene,
// and set our stage size
primaryStage.setTitle("HBox Layout Example");
HBox hb = new HBox();
primaryStage.setScene(new Scene(hb, 400, 300));

// add in some spacing to the hbox and set the background colour
hb.setSpacing(10);
hb.setStyle("-fx-background-color: #0000FF");

// add the 5 buttons to the hbox layout
hb.getChildren().addAll(btn1, btn2, btn3, btn4, btn5);
```

to these

```
// set the title of our stage, create our layout as the root of the scene,
// and set our stage size
primaryStage.setTitle("HBox Layout Example");
VBox vb = new VBox();
primaryStage.setScene(new Scene(vb, 400, 300));

// add in some spacing to the hbox and set the background colour
vb.setSpacing(10);
vb.setStyle("-fx-background-color: #0000FF");

// add the 5 buttons to the hbox layout
```

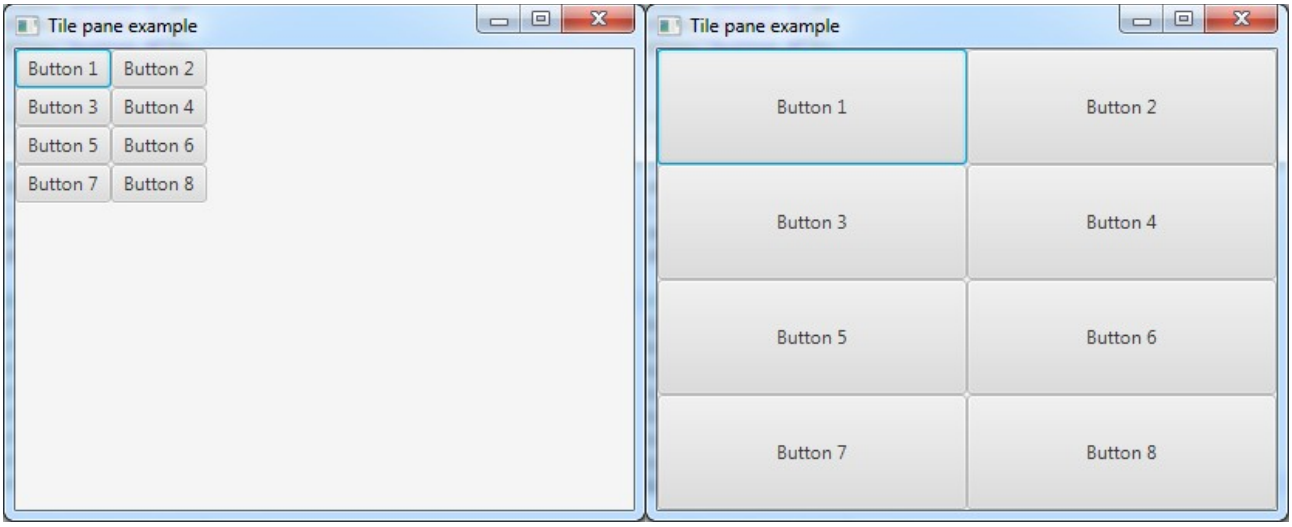
```
vb.getChildren().addAll(btn1, btn2, btn3, btn4, btn5);
```

Explaination:

- As you would expect the way to initialise, style, and set spacing in a VBox is pretty similar to that of a HBox

Example 005: using a GridPane to layout items in a regular grid.

So far we have met three layouts that are regularly used in layouts for the majority of applications. In this example we will meet a fourth called the GridPane which is used for arranging items in a grid with an even space allocated to each item. It is possible to have items that can span multiple cells if necessary. For now we will show the two basic forms of the GridPane as shown in the screenshots below. The screenshot on the left is how the GridPane normally works by restricing itself to the preferred size of the largest widget. The screenshot on the right has modified the code such that all buttons expand to take as much space as possible.



01) start a new java project and create a single source file called Example005.java and give it this shell code

```
// example that shows how the grid pane layout works

// imports
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

// class definiton
public class Example005 extends Application {
	// the init method
	public void init() {
		// initialise all eight buttons
		btn1 = new Button("Button 1");
		btn2 = new Button("Button 2");
		btn3 = new Button("Button 3");
		btn4 = new Button("Button 4");
```

```
                btn5 = new Button("Button 5");
                btn6 = new Button("Button 6");
                btn7 = new Button("Button 7");
                btn8 = new Button("Button 8");
        }

        // the start method
        public void start(Stage primaryStage) {
                // give our stage a title, a grid pane for the scene graph, a window size
                // and set visible
                primaryStage.setTitle("Tile pane example");
                GridPane gp = new GridPane();
                primaryStage.setScene(new Scene(gp, 400, 300));
                primaryStage.show();

                // 03 code

                // 02 code

        }

        // the stop method
        public void stop() {}

        // entry point to our application
        public static void main(String[] args) {
                launch(args);
        }

        // a set of eight buttons that we will use to play with the tile layout
        private Button btn1, btn2, btn3, btn4;
        private Button btn5, btn6, btn7, btn8;
}
```

Explanation:

- as you can see this is pretty much the same shell as the last 3 examples. the only difference here is that we have more buttons and a GridPane as the root of our scene graph

02) replace the comment "02 code" with the following code (good time to compile and run code)

```
// add in the buttons in a 4x2 layout
gp.addRow(0, btn1, btn2);
gp.addRow(1, btn3, btn4);
gp.addRow(2, btn5, btn6);
gp.addRow(3, btn7, btn8);
```

Explaination

- here we are adding the buttons in as a series of rows to the grid pane. Like the addAll() method in previous examples addRow() can take in as many widgets as necessary to construct a row. In order to do this however the first argument must be the index of the row that you are adding. This enables you to add in rows in any

order you wish.

- If you run the code now you will get the screenshot on the left. The GridPane restricts itself to the preferred size of widgets under it's control. If you want a GridPane to expand all widgets to take the maximum space possible then we have to modify the preffered size of all widgets under the grid pane's control. which we do in the next step.

03) replace the comment "03 code" with the following code (good time to compile and run code)

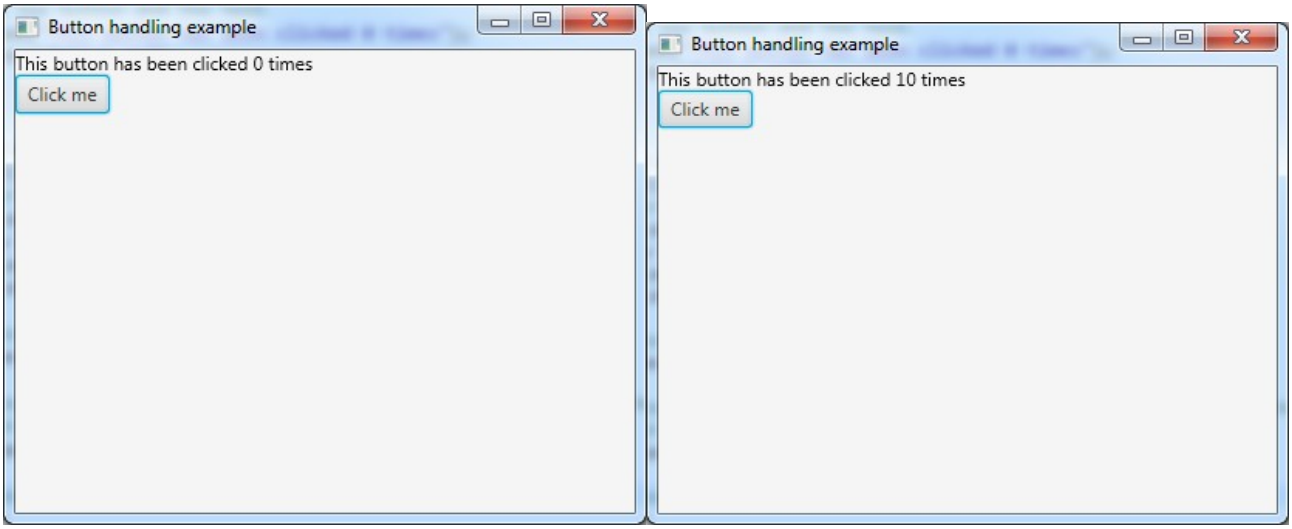
```
// set the buttons to exapand to fill the available space
btn1.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
btn2.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
btn3.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
btn4.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
btn5.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
btn6.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
btn7.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
btn8.setPrefSize(Integer.MAX_VALUE, Integer.MAX_VALUE);
```

Explaination

- this will set the preferred size to the maximum value of an integer. You may have noted in previous examples that I was using doubles but for some reason the Button reacts stragely with doubles but integer max values seem to work fine.
- running this code will give you the screenshot on the right.

Example 006: Introducing the event handling with the button

In the examples you have seen so far we have defined programs that only show you how to use layouts and a couple of basic widgets the Text and the Button. GUI programs are pretty useless if they can't handle input from the user. In this example we will see how the event handling system works by getting the button to update a Text widget everytime it is clicked. The screenshots below show the application starting (on the left) and after the button has been clicked 10 times (on the right)



01) start with a new Java project and create a single source file Example006.java and give it the following shell code

```
// example to show a button with some basic event handling

// imports
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class Example006 extends Application {
	// the init method
	public void init() {
		// initialise the button and the text
		label = new Label("This button has been clicked 0 times");
		btn = new Button("Click me");
		clicks = 0;
	}

	// the start method
	public void start(Stage primaryStage) {
```

```
                // create a primary stage with a vbox layout, title, and size
                primaryStage.setTitle("Button handling example");
                VBox vb = new VBox();
                primaryStage.setScene(new Scene(vb, 400, 300));
                primaryStage.show();

                // add the text and the button to the vertical box layout
                vb.getChildren().addAll(label, btn);

                // 02 code
        }

        // the stop method
        public void stop() {}

        // the entry point of our program
        public static void main(String[] args) {
                launch(args);
        }

        // private fields for a button and a text and an integer to keep a count of the total
number of clicks
        private Label label;
        private Button btn;
        private int clicks;

}
```

Explanation:

- the init method is not just there to initialise JavaFX components. it can also be used to initialise anything else in your code. In this case we initialise a counter before it will ever get used.
- The label and button are added to a vbox layout to keep things simple

02) add this in place of the comment "02 code" (good time to compile and run code)

```
// add a listener to the button to react to a click by the user. this should increment
// the number by 1 and refresh the screen
btn.setOnAction(new EventHandler<ActionEvent>() {

        // must override this method for an action event event handler
        @Override
        public void handle(ActionEvent event) {
                clicks++;
                label.setText("This button has been clicked " + clicks + " times");
        }
});
```

Explaination

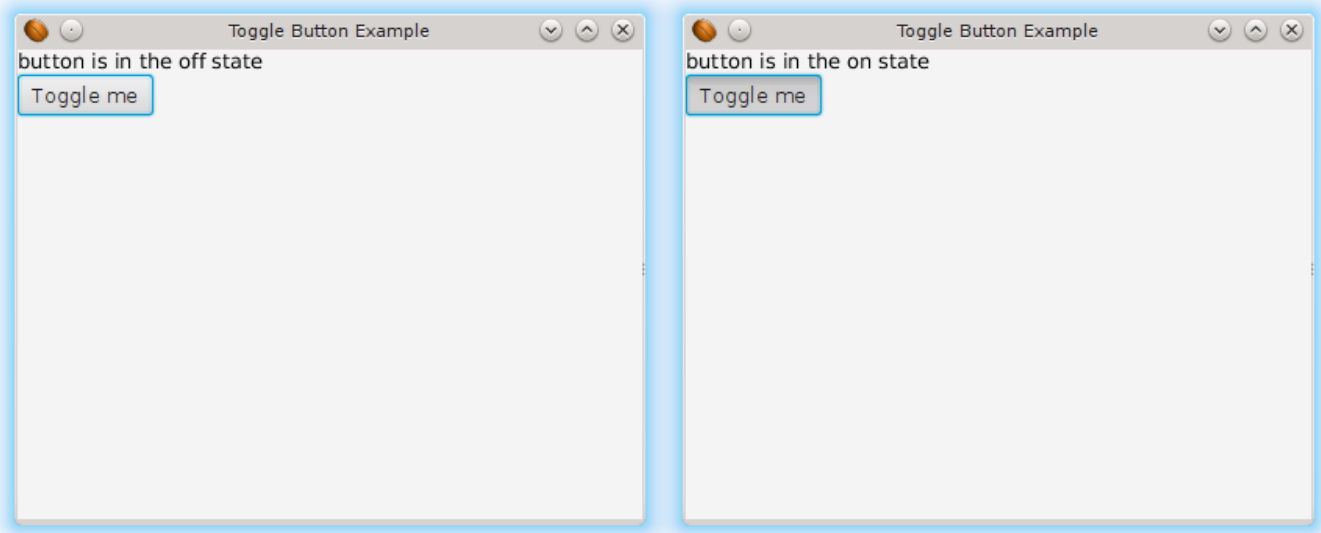
- Events are handled through the use of the generic EventHandler class. This is a generic class because there are many different types of event that can be handled in a JavaFX application. In this case (and indeed the majority of cases) we will

configure this handler to handle an ActionEvent as these are the events that are produced whenever a Button is clicked.

- The setOnAction() method sets an ActionEvent handler on a button. In this case we are generating an anonymous class which will only ever be used by this button. Thus it is not necessary to break it into a seperate Java file. while this makes your source file much bigger it is possible to break then into seperate methods to make it easier to manage (in later examples with more events I will do this).
- All EventHandler implementations must override the handle method. Please note that the class of the parameter event must match the class in <> at the declaration of the EventHandler anonymous class.
- In the method above we simply increment the number of clicks and we update the text on the Label widget to reflect the change.

Example007: introducing the ToggleButton and its associated event handling

In this example we show the ToggleButton which is like the button you have seen before but it has both a pressed in and a pressed out state. The two screenshots below show the result you get after building this example.



01) start with a new java project. Create a single source file called Example007.java and give it the following shell code

```
// simple javafx program showcasing the toggle button

// imports
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.ToggleButton;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.stage.Stage;

// class definition
public class Example007 extends Application {
	// overridden version of the init method
	public void init() {
		// initialise the button and the text
		tbtn_button = new ToggleButton("Toggle me");
		label_display = new Label("button is in the off state");
	}
```

```
	// overridden version of the start method
	public void start(Stage primaryStage) {
		// set the primary stage to have a title size and vbox layout
		primaryStage.setTitle("Toggle Button Example");
		VBox vb = new VBox();
		primaryStage.setScene(new Scene(vb, 400, 300));
		primaryStage.show();

		// add the text and the button to the layout
		vb.getChildren().addAll(label_display, tbtn_button);

		// 02 code goes here
	}

	// overridden version of the stop method
	public void stop() {

	}

	// entry point of our program
	public static void main(String[] args) {
		launch(args);
	}

	// private fields of a Text and a ToggleButton
	private Label label_display;
	private ToggleButton tbtn_button;
}
```

Explanation:

- This is pretty much the same shell as before however instead of using the Button class we are using the ToggleButton class which is instantiated in the same way as a Button by providing a string of text in the constructor to set on the face of the button itself.

02) replace the comment for 02 code with the following code

```
// add an event listener to the button to update the text
tbtn_button.setOnAction(new EventHandler<ActionEvent>() {
	// overridden version of the handle method to catch events
	@Override
	public void handle(ActionEvent event) {
		// change the text depending on the state of the button
		if(tbtn_button.isSelected())
			label_display.setText("button is in the on state");
		else
			label_display.setText("button is in the off state");
	}
});
```

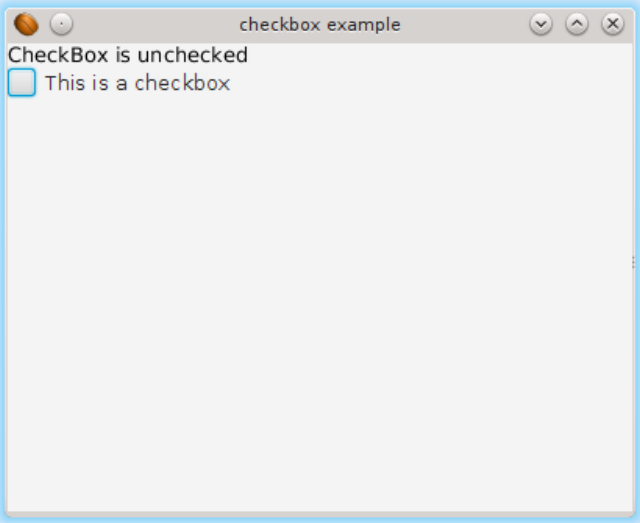
Explanation:

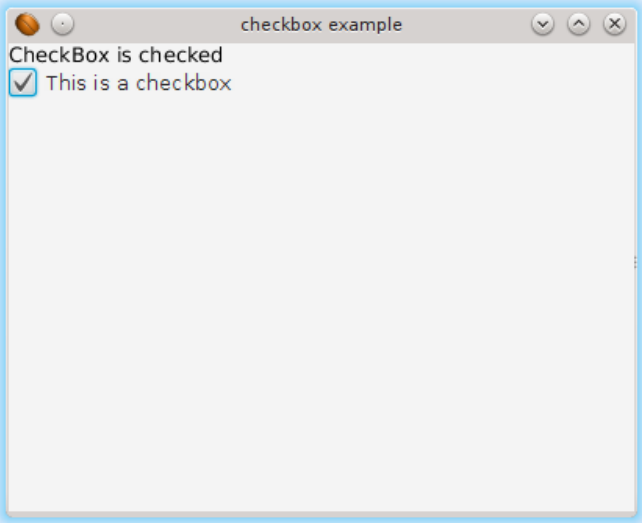
- like the Button class events are dealt with using the generic event handler class. So the EventHandler uses the ActionEvent class as before and the handle method is

overridden. In the handle method if you wish to query the state of the toggle button you use the isSelected() method. This will return true if the button is pushed in or false if the button is pushed out. We use this to modify the state of our Text.

Example008: Introducing the checkbox and its event handling.

In this example we will continue in a similar vein to previous examples by presenting a single control and how its event handling works. In this example we will explore the checkbox which is a more common variant of the toggle button. Checkboxes are binary in nature and exist in a checked state or unchecked state, both of which are shown below.





01) start with a new java project, create a single source file called Example008.java and give it the following shell code

```
// simple example that showcases the checkbox

// imports necessary to make this project work
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.stage.Stage;

// class definition
public class Example008 extends Application {
	// the overridden init method
	public void init() {
		// create the text and the checkbox
		label_display = new Label("CheckBox is unchecked");
		cb_checkbox = new CheckBox("This is a checkbox");
	}

	// overridden start method
	public void start(Stage primaryStage) {
		// create a scene of a vbox layout, set the window size and title
```

```
too
            primaryStage.setTitle("checkbox example");
            VBox vb = new VBox();
            primaryStage.setScene(new Scene(vb, 400, 300));
            primaryStage.show();

            // add the text and the checkbox to the layout
            vb.getChildren().addAll(label_display, cb_checkbox);

            // 02 code goes here

      }

      // overridden stop method
      public void stop() {

      }

      // entry point to our program
      public static void main(String[] args) {
            // launches our application
            launch(args);
      }

      // private fields for the text and the checkbox
      private Label label_display;
      private CheckBox cb_checkbox;
}
```

Explanation

- this is pretty much the same as for the toggle button but instead it is using a checkbox

02) replace the comment for 02 code with the following code.

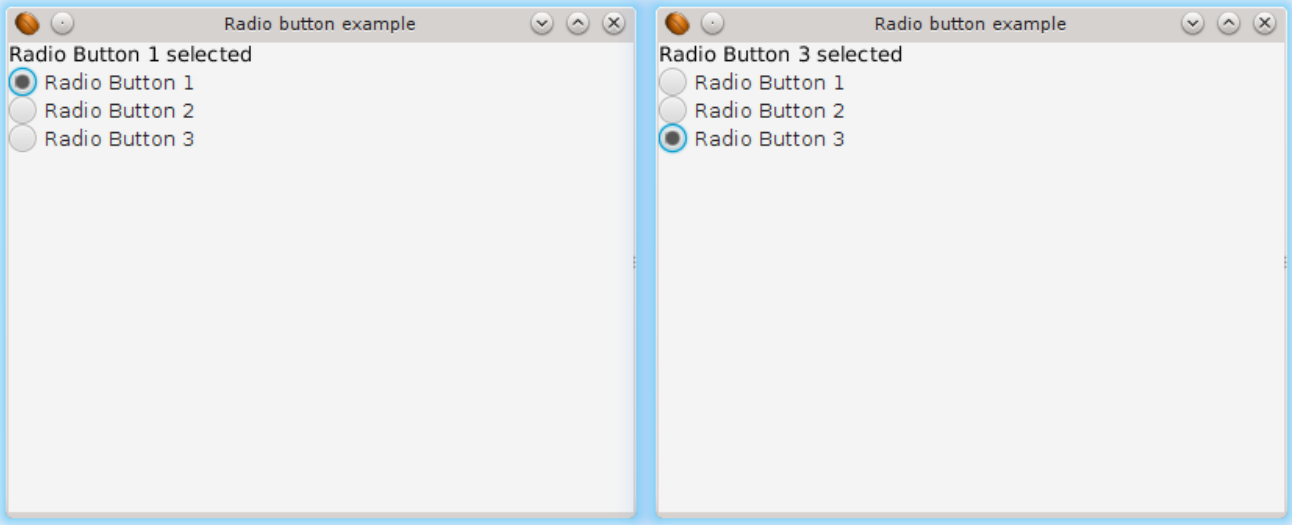
```
// add an event listener to the checkbox that will modify the text when it
// is clicked
cb_checkbox.setOnAction(new EventHandler<ActionEvent>() {
      // override the handle method to detect and react to events
      @Override
      public void handle(ActionEvent event) {
            // change the text depending on whether the checkbox is checked
            if(cb_checkbox.isSelected())
                  label_display.setText("CheckBox is checked");
            else
                  label_display.setText("CheckBox is unchecked");
      }
});
```

Explanation

- this is pretty much the same as for the toggle button but instead it is using a checkbox

Example009: Introducing the RadioButton class and its event handling

In this example we will introduce the RadioButton class. RadioButtons are used to force a choice among a number of options. Thus when the user changes their mind the previous option will automatically be deselected i.e. it forces a mutal exclusion choice. An example of this is seen in the screenshots below.



01) start with a new Java project. Create a single source file called Example009.java and give it the following shell code.

```
// simple example that showcases the radiobutton class

// imports
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.VBox;
import javafx.scene.control.Label;
import javafx.stage.Stage;

// class definition
public class Example009 extends Application {
	// overridden init method
	public void init() {
		// 02 code goes here
	}

	// overridden start method
	public void start(Stage primaryStage) {
		// create a window with a vbox layout, set its size and title
```

```
            primaryStage.setTitle("Radio button example");
            VBox vb = new VBox();
            primaryStage.setScene(new Scene(vb, 400, 300));
            primaryStage.show();

            // add the text and the radio buttons to our layout
            vb.getChildren().addAll(label_display, rb1, rb2, rb3);

            // 03 code goes here

            // 04 code goes here
      }

      // overridden stop method
      public void stop() {

      }

      // entry point to our program
      public static void main(String[] args) {
            // launch our application
            launch(args);
      }

      // private fields that include radio buttons, a toggle group for grouping
      // the radio buttons and a text for showing event handling
      private RadioButton rb1, rb2, rb3;
      private ToggleGroup tg_radiobutton_group;
      private Label label_display;
}
```

Explaination:

- If you look at the private fields we have the text and radiobutton classes as expected. However there is an instance of a ToggleGroup too. The ToggleGroup is required to force the mutual exclusion on the RadioButton. If you omit the ToggleGroup from radio button setup it is possible to select more than radiobutton at a given time.

02) replace the comment for 02 code with the following code

```
// initialise the radio buttons, toggle group and Text
tg_radiobutton_group = new ToggleGroup();
label_display = new Label("Radio Button 1 selected");
rb1 = new RadioButton("Radio Button 1");
rb2 = new RadioButton("Radio Button 2");
rb3 = new RadioButton("Radio Button 3");

// add the radio buttons to the toggle group
rb1.setToggleGroup(tg_radiobutton_group);
rb2.setToggleGroup(tg_radiobutton_group);
rb3.setToggleGroup(tg_radiobutton_group);

// set the first radio button to be selected by default
rb1.setSelected(true);
```

Explaination:

- Radio buttons require more setup than Buttons or CheckBoxes. In this case the first block of code is creating our ToggleGroup for our RadioButtons and a text display to show a reaction to events. Note that the text passed into the constructor for the radio button will appear directly to the right of the radio button in the GUI.
- In the second block of code we add all three buttons to the toggle group to force mutual exclusion. You are not required to do anything extra as the toggle group will handle the mutual exclusion internally.
- The last line is defaulting our radiobuttons to have the first button selected. It is a good idea to do this for the sake of consistency as if no default is set then all radio buttons will be unselected until the user makes the first click. After this normal radio button behaviour will resume.

03) replace the comment for 03 code with the following code

```
// add an event listener to the first radio button
rb1.setOnAction(new EventHandler<ActionEvent>() {
	// override this method to handle events
	public void handle(ActionEvent event) {
		label_display.setText("Radio Button 1 selected");
	}
});
```

Explaination:

- similar event handling to the widgets we have seen so far. Note that we dont check for selection here as a press on a radio button automatically implies selection.

04) replace the comment for 04 code with the following code

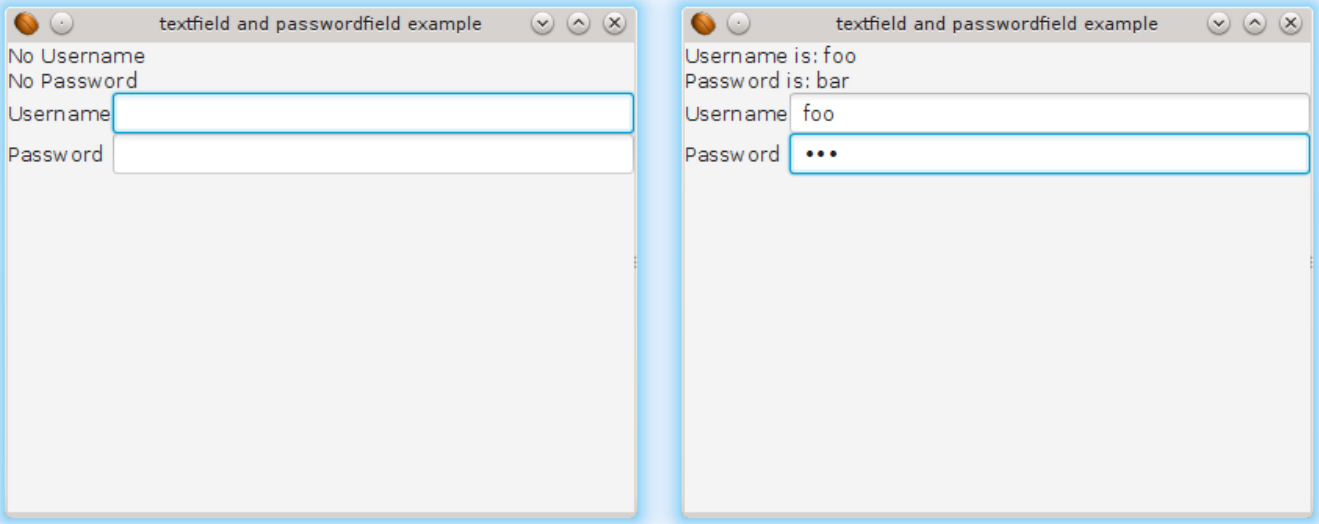
```
// add an event listener to the second radio button
rb2.setOnAction(new EventHandler<ActionEvent>() {
	// override this method to handle events
	public void handle(ActionEvent event) {
		label_display.setText("Radio Button 2 selected");
	}
});

// add an event listener to the third radio button
rb3.setOnAction(new EventHandler<ActionEvent>() {
	// override this method to handle events
	public void handle(ActionEvent event) {
		label_display.setText("Radio Button 3 selected");
	}
});
```

Explaination:

- same event handling for the other radio buttons

Example010: introducing the TextField and PasswordField classes and their event handling

In this example we will show the TextField class and the PasswordField class together along with their associated event handling. TextFields and PasswordFields are used to obtain single line input from a user. As you can guess the text field will show the user exactly what they have typed whereas the password field will show a generic symbol for each character to show that the user has entered characters but will not specify which. In this example you will generate the two screenshots you see below.



01) start with a new java project create a single source file called Example010.java and give it the following shell code.

```
// simple example that shows both the text field and the password field

// imports
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.PasswordField;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example010 extends Application {
      // overridden init method
      public void init() {
            // code for 02
```

```
            // code for 03

            // code for 04

      }

      // overridden start method
      public void start(Stage primaryStage) {
            // create a window with a vbox layout, set its size and title
            primaryStage.setTitle("textfield and passwordfield example");
            VBox vb = new VBox();
            vb.setFillWidth(true);
            primaryStage.setScene(new Scene(vb, 400, 300));
            primaryStage.show();

            // add all the elements to our layout
            vb.getChildren().addAll(label_username, label_password,
gp_userpass);
      }

      // overridden stop method
      public void stop() {

      }

      // entry point to our program
      public static void main(String[] args) {
            // launch our application
            launch(args);
      }

      // private fields that include texts, text field and password field
      private Label label_username, label_password, label_ulabel, label_plabel;
      private TextField tf_username;
      private PasswordField pf_password;
      private GridPane gp_userpass;
}
```

Explaination:

- The TextField class is used when a text field is required and the PasswordField class is used whenever a password field is required.
- You may notice that we have a grid pane here. The reason we have this is to put the username and password along with labels for both in a neat 2x2 layout with a label with the words "Username" in the top left, a label with the word "Password in the bottom left", a text field in the top right, and a password field in the bottom right.

02) replace the code for 02 comment with the following code

```
      // initialise all of our controls
      label_username = new Label("No Username");
      label_password = new Label("No Password");
      label_ulabel = new Label("Username");
      label_plabel = new Label("Password");
```

```
        tf_username = new TextField();
        pf_password = new PasswordField();
        gp_userpass = new GridPane();

        // put the texts and fields together in a grid pane layout
        gp_userpass.add(label_ulabel, 0, 0);
        gp_userpass.add(label_plabel, 0, 1);
        gp_userpass.add(tf_username, 1, 0);
        gp_userpass.add(pf_password, 1, 1);
        GridPane.setHgrow(tf_username, Priority.ALWAYS);
        GridPane.setHgrow(pf_password, Priority.ALWAYS);
```

Explaination:

- The first block of code is standard initialisation which we have seen before. The second part however has a few differences. If you remember in the previous examples with the grid pane we added elements on a row by row basis. Here we are adding in elements individually by specifiing a row number and the column number for where that element should be placed.
- The grid pane will then figure out how many elements it has in each direction and will generate the appropriate layout.
- The last two lines of code however are there to ensure that the grid pane will take advantage of any extra horizontal space allocated to it. If you omit these two lines you will see a text field and a password field that goes no further than 2/3rds of the way across the window. No matter how much you resize the window it will not take any extra space.
- setHGrow is a static member of the GridPane class that enables you to allocate priority for extra space. In these two lines we are stating that should extra space become available then space should always be allocated to the text field and the password field. This makes sense because there is no point in allocating extra space to the labels on the left as they will not make any use of that space.

03) replace the comment for 03 code with this code

```
// add an event handler to the text field
tf_username.setOnAction(new EventHandler<ActionEvent>() {
        // override this method to handle events
        @Override
        public void handle(ActionEvent event) {
                // as the enter key has been pressed here we will update the label
                // for the username to show what was entered
                label_username.setText("Username is: " + tf_username.getText());
        }
});
```

Explaination:

- This looks very similar to event handling for buttons and checkboxes. However the context of an action event is different. In this case an ActionEvent is triggered when the return key is pressed while the cursor is in the TextField.

- Please note that it is entirely your responsibility to validate all text that is entered in text fields. A common way that apps are insecure and exhibit bugs is that they do not validate their user input.
- Here as soon as the return key is pressed the text will be taken from the TextField and will be used to update the label with the username.

04) replace the comment for 04 code with this code

```
pf_password.setOnAction(new EventHandler<ActionEvent>() {
	// override this method to handle events
	@Override
	public void handle(ActionEvent event) {
		// as the enter key has been pressed here we will update the label
		// for the password to show what was entered
		label_password.setText("Password is: " + pf_password.getText());
	}
});
```

Explaination:

- pretty much the same event handling as for the TextField

Example 011: Introducing the TextArea and its event handling

Another useful control that is provided by default with the JavaFX toolkit is the TextArea. This is used for editing multiple lines of text for more complex user input. Something like this would generally be used for text editors. In this example we will show how to initialse and set text areas but we will also show how to handle events from that text area itself.

01) start with a new java project. Create a single source file called Example011.java and give it the following shell code.

```
// simple example to show how the text area works and its event handling

// imports
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example011 extends Application {
	// the overridden import method
	public void init() {
		// 02 code goes here

		// 03 code goes here
	}

	// the overridden start method
	public void start(Stage primaryStage) {
		// set a title on the primary stage, a size and a scene
		primaryStage.setTitle("TextArea example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// the overridden stop method
	public void stop() {

	}

	// entry point into our program for launching JavaFX
	public static void main(String[] args) {
		launch(args);
	}

	// private fields consisting of a stack pane and a textarea
	private VBox vb_mainlayout;
	private TextArea ta_textarea, ta_response;
}
```

02) replace the comment for 02 code with the following code

```
// initialise our layouts and widgets
vb_mainlayout = new VBox();
ta_textarea = new TextArea();
ta_response = new TextArea();

// attach the areas to the pane and disable editing of the response
vb_mainlayout.getChildren().addAll(ta_textarea, ta_response);
ta_response.setEditable(false);
```

Explaination:

- as before the first block of code initialises all of our widgets and layouts.
- The second block of code adds all of our widgets to our layout. However the second line in this block renders the second of our text areas uneditable. This can be useful if you wish to have a text area that will display console or logging output that you do not wish your user to have edit.

03) replace the comment for 03 code with the following code (good time to compile and run)

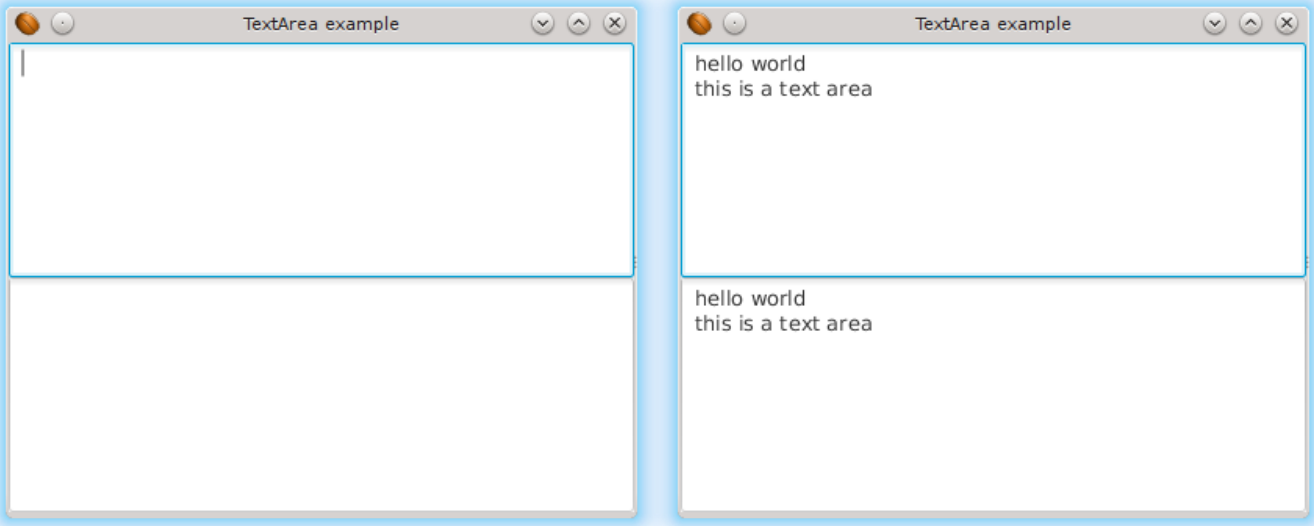
```
// add in a handler to the event
ta_textarea.textProperty().addListener(new ChangeListener<String>() {
      // method that must be overridden to listen for changes in the text area
      @Override
      public void changed(final ObservableValue<? extends String> observable,
final String oldValue, final String newValue) {
            ta_response.setText(newValue);
      }
});
```

Explaination:

- Here is how we add a listener to the text area. Note that this is a much different format to the listeners that you have seen so far. This is because the text area requires you to listen to the textProperty directly in order to pick up changes in the text.
- The text property accepts multiple listeners and they are added using the addListener() method. This method accepts objects that implement the ChangeListener interface. In this case because we are dealing with strings we modify this class to work with strings.
- Take careful note of the signature of the changed method. Particularly the three arguments that are provided. All arguments are stated to be final because the documentation explicitly states that it is bad practice to modify the observed value in this method.
- The second and third arguements denote the previous and current values of the textarea in case you wish to take note of the differences between the two. The first value we will ignore but I will explain a bit of java syntax that you have not seen before.
    - <? extends String> means that there may be subclasses or subtypes that are

handled by the ObservableValue class but we don't know their exact type however what this states is that regardless of the subtype we know that they are extensions of the String class meaning that whatever object we get back we can call methods on the observable value that correspond to the String class safely.

- In this particular case we ask the listener to update the uneditable text field with any new text that is entered. What you will see when you run this program is that the text will update on every character that is entered.
- this code will get you the following screenshots. The screenshot on the left is the initial state of the program while the screenshot on the right shows the state after some text has been entered into the top textarea.



Example012: introducing the combobox and its event handling

Another useful component that is provided by default with javafx is the combobox. The combobox is a generic type that gives the user a dropdown list with a set of choices. A good place to use this is where you would ideally use radio buttons but you are restriced on space. In this example we will show how to initialise a combobox to use strings, populate the combobox and also show how to handle events generated from the combobox.

01) start with a new java project. Generate a single source file called Example012.java and add the following shell code into it

```
// simple example to show how the combobox works and how
// to listen for and react to events

// imports
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example012 extends Application {
	// overridden init method
	@Override
	public void init() {
		// 02 code goes here

		// 03 code goes here
	}

	// overridden start method
	@Override
	public void start(Stage primaryStage) {
		// set a title and size on our window and add the main layout
		// to the root of the scene graph
		primaryStage.setTitle("ComboBox Example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden stop method
	@Override
	public void stop() {

	}

	// entry point into our program which launches our JavaFX
	// application
	public static void main(String[] args) {
```

```
            launch(args);
      }

      // private fields of this class
      private Label lbl_display;
      private VBox vb_mainlayout;
      private ComboBox<String> cmb_combobox;
}
```

Explanation:

- Note how the combobox is a generic type. It is possible to add your own custom types into the combobox but this will require custom cell renderers that we will not go into detail now about. You can use any of the default number types here instead without issue.

02) replace the comment for 02 code with the following code

```
// initialise all of our widgets and layouts
vb_mainlayout = new VBox();
lbl_display = new Label("ComboBox: No current selection");
cmb_combobox = new ComboBox<String>();
cmb_combobox.getItems().addAll("This is choice 1", "This is choice 2", "This is
choice 3");
```

Explanation:

- the first two lines are standard but the last two lines are how we initialise a combo box.
- The first line initialises a combobox for handling String objects. The second line populates the combobox with a number of items by adding them to the combobox's internal list of items. You may add as many items as you wish.

03) replace the comment for 03 code with the following code (good time to compile and run)

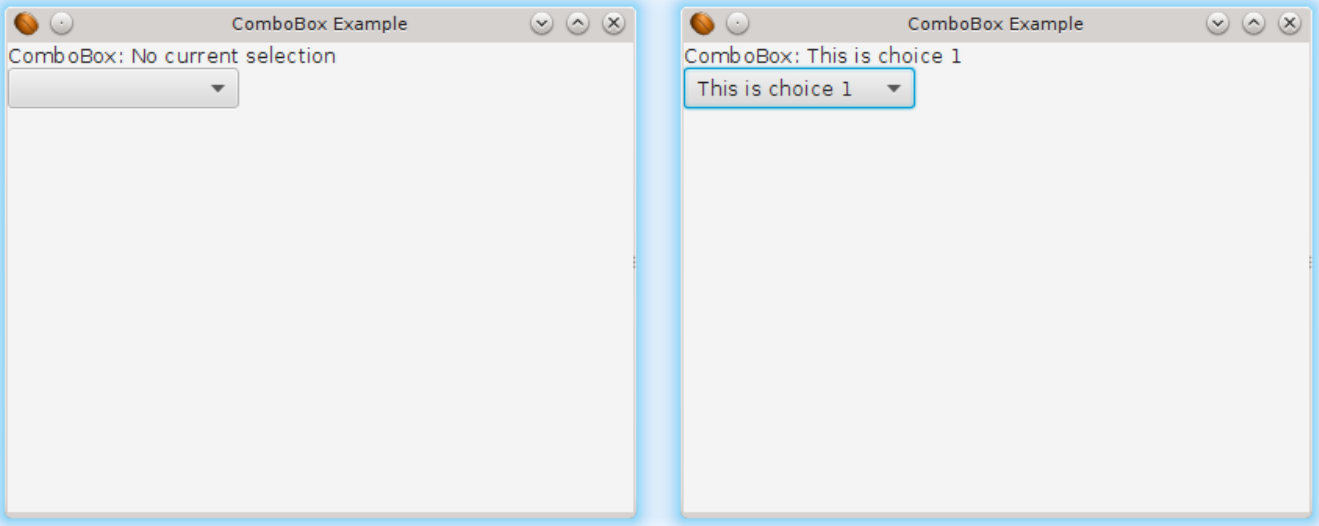
```
// add all the children to the layout and set an event listener
// on the combobox
vb_mainlayout.getChildren().addAll(lbl_display, cmb_combobox);
cmb_combobox.setOnAction(new EventHandler<ActionEvent>() {
      // overridden handle method
      @Override
      public void handle(ActionEvent event) {
            // change the text depending on what choice was made
            lbl_display.setText("ComboBox: " + cmb_combobox.getValue());
      }
});
```

Explanation:

- This is how add an event handler to the ComboBox. This uses the same EventHandler and handle function that you saw before. However handling a combobox is a bit more complex. In many cases you will do different actions depending on the value in a combobox so in general you will see an if-else-if ladder

inside this method for a combobox.

- Here we get the currently selected value from the combobox by using the getValue() method. If a combobox has been set to use String objects then this will return a String object.
- Running the code at this point should yield you the screenshots that you see below. The screenshot on the left shows the initial state of the combobox where no item is selected, while the screenshot on the right shows the combobox after a value has been selected
    - if you wish to set a default value on the combobox there is a method setValue() provided. In this case you would have to provide the string that you wish to have as default to the setValue() method to set it as default.



Example013: introducing the slider and its event handling

Sliders are another useful component provided by JavaFX. A slider is used to get a user to input a numerical value that has a well defined upper bound and lower bound. It is recommended that you use sliders in your applications as they provide you with a range of expect values and perform automatic validation of user input. In this example we will show how to setup a Slider and also observe the value of the slider to detect changes.

01) start with a a new java project. Generate a single source file called Example013.java and give it the following shell code

```
// simple javafx example showing the slider and associated
// event handling

// imports
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example013 extends Application {
	// overridden init method
	@Override
	public void init() {
		// 02 code here

		// 03 code here
		sli_slider.setShowTickMarks(true);
		sli_slider.setShowTickLabels(true);
		sli_slider.setMajorTickUnit(1);
		sli_slider.setBlockIncrement(0.5);

		// 04 code here
	}

	// overridden start method
	@Override
	public void start(Stage primaryStage) {
		// set a size, title, and scene on the window and show it
		primaryStage.setTitle("Slider Example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden stop method
	@Override
	public void stop() {

	}
```

```
	// entry point into our program that launches our JavaFX
	// application
	public static void main(String[] args) {
		launch(args);
	}

	// private fields of the class
	private Label lbl_display;
	private Slider sli_slider;
	private VBox vb_mainlayout;
}
```

02) replace the comment for 02 code with the following code

```
	// initialise all of our layouts and widgets
	lbl_display = new Label("slider value set to: 5.0");
	sli_slider = new Slider(0, 10, 5);
	vb_mainlayout = new VBox();
	vb_mainlayout.getChildren().addAll(lbl_display, sli_slider);`
```

Explaination:

- The constructor of the slider expects three arguments all in Double form. The first two arguments are the lower bound and upper bound of the slider. The final value is the initial value that the slider is set to. In this case we have a slider that can select any value between [0,10] but it starts at 5.

03) replace the comment for 03 code with the following code

```
	// customise the slider a little bit
	sli_slider.setShowTickMarks(true);
	sli_slider.setShowTickLabels(true);
	sli_slider.setMajorTickUnit(1);
	sli_slider.setBlockIncrement(0.5);
```

Explaination:

- Normally when you add in a slider without doing this configuration you will only see the tick mark and number for the lower bound but will have no reference for other values.
- The first line here enables tick marks for all other values. The second line adds labels to each major tick line. The labels are determined by the third line which sets the major ticks (the longer of the ticks that you see) to be every 1 unit. Thus the ticks at values {0, 1, 2, ... , 10} will all have a label indicating what their value is.
- The final line sets the block increment which is used whenever a user is using the keyboard to manipulate the value of the slider. This increment is set to 0.5 meaning that if the right arrow or left arrow on the keyboard is pressed then it will increase or decrease by 0.5 respectively.

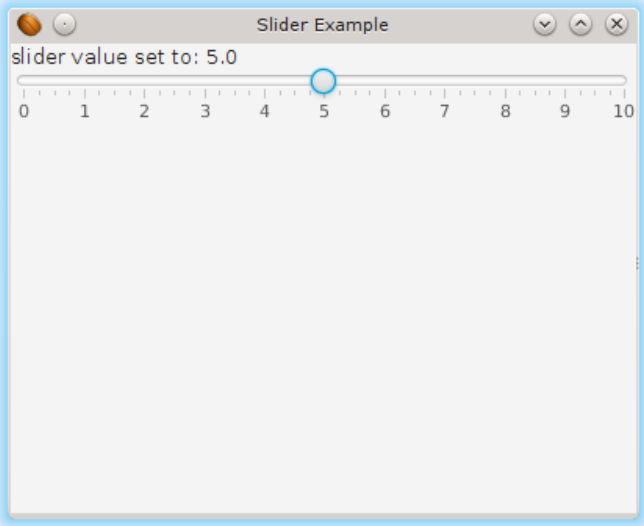
04) replace the comment for 04 code with the following code

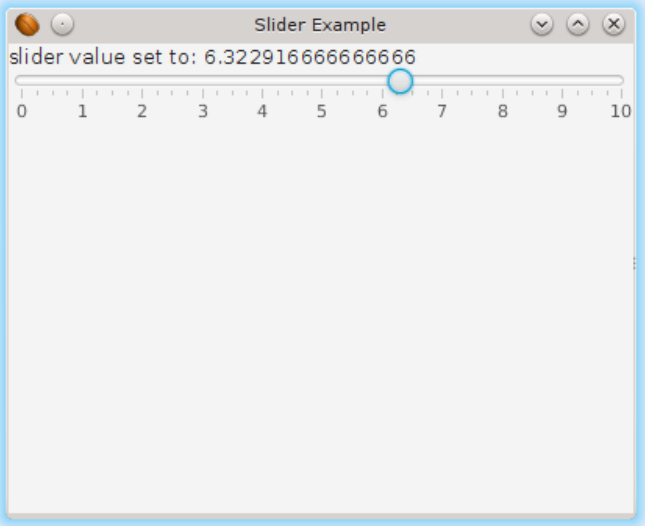
```
// add an event listener into the slider
sli_slider.valueProperty().addListener(new ChangeListener<Number>() {
```

```
        public void changed(final ObservableValue<? extends Number> observable,
final Number oldValue, final Number newValue) {
                lbl_display.setText("slider value set to: " + newValue);
        }
});
```

Explaination:

- slider event handling is done in a similar manner to that of the text area in that we need to add a listener to a property maintained by the slider. In this case we have to add a listener to the valueProperty. We use a change listener as before but instead of using a String class for the Listener we use a Number class instead. As before we can only guarentee that the value will be based on the Number class but we don't know what type of number it will be. However, sliders generally use doubles so you can be sure that you will have a floating point number of some description.
- Here we use the number to update the text with the new value that was selected by the user.
- Running this code should net you the screenshots that you see below. The left screenshot shows the initial state of the application, while the right screenshot shows the application after the user has changed the value of the slider by dragging it.





Example014: introducing the menu bar and the associated menu items and their event handling.

Most useful desktop applications will have a menubar at the top of their window which hides away multiple options but makes them easy to access. This is done to keep the majority of window space free for showing the main task that the applicaton is to perform. In this example we will show how to add a menu bar to an application and also setup menu items. To finish off we will show how to react to events from these menu items.

01) start with a new java project. Create a single source file called Example014.java and add the following shell code to it

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.CheckMenuItem;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.control.RadioMenuItem;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// simple example showcasing menus and their event handling

// imports

// class definition
public class Example014 extends Application {
	// overridden init method
	@Override
	public void init() {
		// 02 code goes here

		// 03 code goes here

		// 04 code goes here

		// 05 code goes here

	}

	// overridden start method
	public void start(Stage primaryStage) {
		// set the title, size and show the window
		primaryStage.setTitle("menus example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();


	}
```

```
	// overridden stop method
	public void stop() {

	}

	// entry point to our program to launch our JavaFx application
	public static void main(String[] args) {
		launch(args);
	}

	// private fields of the class
	private VBox vb_mainlayout;
	private Label lbl_display;
	private MenuBar mb_menubar;
	private Menu menu_file, menu_help;
	private MenuItem mi_hello, mi_quit;
	private RadioMenuItem rmi_rb1, rmi_rb2;
	private CheckMenuItem cbmi_cb1, cbmi_cb2;
	private ToggleGroup tg_rmi;

}
```

Explaination:

- Note that their are a lot of fields declared here. There are multiple classes as well thus we will explain what each menu class represents.
- The MenuBar is the class that holds all menus and is the bar that should be displayed at the top of the application. You should only have one menubar per window.
- The Menu class implements the logic for displaying a menu and for containing menu items. Examples of commonly found menus in applications are the File menu and the Help menu. We will implement these menus by using the Menu class as a container.
- The MenuItem class represents a standard menu item. This should be treated the same as the Button class. i.e. when it is clicked a single action should be performed.
- The RadioMenuItem is a menu item that behaves exactly like the RadioButton class. It is used to represent mutual exclusion, therefore like RadioButtons they need to be assigned to a ToggleGroup to force the mutual exclusion.
- The CheckMenuItem is a menu item that behaves exactly like the CheckBox class. It is used to represent an option that can be switched on or off.

02) replace the comment for 02 code with the following code

```
	// initialise our widgets and layouts
	vb_mainlayout = new VBox();
	lbl_display = new Label("No menu item selected");
	mb_menubar = new MenuBar();
```

```
        menu_file = new Menu("File");
        menu_help = new Menu("Help");
        mb_menubar.getMenus().addAll(menu_file, menu_help);
        vb_mainlayout.getChildren().addAll(mb_menubar, lbl_display);
```

Explaination:

- If you wish to add a menu to your application this forces your applicaton to have either a border pane or a vbox as your root layout as good GUI design demands that the menubar be at the top and take the full width of the GUI window. These two layouts will implement this for you without effort.
- All menus should be initialised with a string. This string will be displayed on the GUI as the name for the menu.
- A menu bar maintains a list of menu objects this list can be obtained by calling the getMenus() method and to add menus there is an addAll function that can be used to add menus to the menu bar.

03) replace the comment for 03 code with the following code

```
// add in the menu items and check menu items to the file menu to the menus
mi_hello = new MenuItem("Hello");
mi_quit = new MenuItem("Quit");
cbmi_cb1 = new CheckMenuItem("check box 1");
cbmi_cb2 = new CheckMenuItem("check box 2");
menu_file.getItems().addAll(mi_hello, mi_quit, cbmi_cb1, cbmi_cb2);
```

Explaination:

- here we are initialising MenuItems and CheckMenuItems and adding them to the file menu.
- Each menu maintains a list of all added MenuItems which can be obtained by calling the getItems() method. To add items to this list you can use the addAll() method.

04) replace the comment for 04 code with the following code

```
// add the radio buttons menu items to a toggle group and add them to the
// help menu
tg_rmi = new ToggleGroup();
rmi_rb1 = new RadioMenuItem("Radio button 1");
rmi_rb2 = new RadioMenuItem("Radio button 2");
rmi_rb1.setToggleGroup(tg_rmi);
rmi_rb2.setToggleGroup(tg_rmi);
menu_help.getItems().addAll(rmi_rb1, rmi_rb2);
```

Explaination:

- Here we perform a similar job to the 03 code above for adding RadioMenuItems into the help menu. Note that because we have radio items here we need to set a toggle group on the RadioMenuItems to ensure that mutual exclusion works.

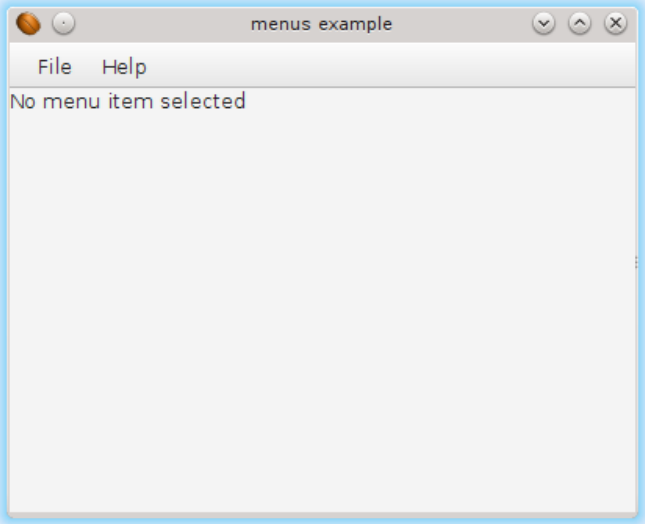
05) replace the comment for 05 code with the following code (good place to compile and

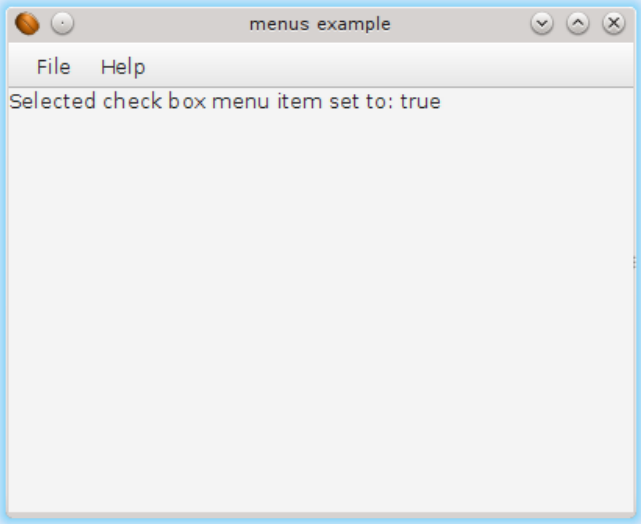
run)

```
// add in an event listener to a menu item, a check menu item and a radio menu
item
mi_hello.setOnAction(new EventHandler<ActionEvent>() {
	// overridden method to handle an event for this menu item
	@Override
	public void handle(ActionEvent event) {
		lbl_display.setText("Selected hello menu item");
	}
});
cbmi_cb1.setOnAction(new EventHandler<ActionEvent>() {
	// overridden method to handle and event for this check menu item
	@Override
	public void handle(ActionEvent event) {
		lbl_display.setText("Selected check box menu item set to: " +
cbmi_cb1.isSelected());
	}
});
rmi_rb1.setOnAction(new EventHandler<ActionEvent>() {
	// overridden method to handle an event for this radio menu item
	@Override
	public void handle(ActionEvent event) {
		lbl_display.setText("Selected radio menu item 1");
	}
});
```

Explaination:

- here we show how to add an event listener to a MenuItem, CheckMenuItem and a RadioMenuItem. Note that we have not added listeners to all menu items.
- These three menu item types all have the setOnAction() method that take in an EventHandler object. This works in exactly the same way as the Button and the RadioButton class examples that you have already seen. In the case of the CheckMenuItem you will require the isSelected() method to determine if the menu item has a check present. Thus in the majority of cases your handle method will have an if else statement in it.
- Running the code will get you the screenshots you see below. In the first shot we see the initial state of the application. Whereas in the second shot we see the application after the first of the CheckMenuItems has been clicked.


menus example
File
Help
No menu item selected


menus example
File
Help
Selected check box menu item set to: true

Example015: Introducing the progress bar.

Sometimes in applications you will want to indicate the progress of an action as it is being performed. This is to give the user visual feedback on the operation in progress. It is generally recommended that you use progress bars for actions that have a definite start and end and also where the action takes a significant period of time. In this example we will show how to initialise and use a progress bar with help from a slider

01) start a new java project, generate a single source file called Example015.java and give it the following shell code

```
// simple example that shows how the progress bar works

// imports
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.Slider;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example015 extends Application {
	// overridden init method
	public void init() {
		// 02 code goes here

		// 03 code goes here

		// 04 code goes here
	}

	// overridden start method
	public void start(Stage primaryStage) {
		// set a title, size and scene
		primaryStage.setTitle("Progress Bar example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden stop method
	public void stop() {

	}

	// entry point into our program that will launch our javafx example
	public static void main(String[] args) {
		launch(args);
	}

	// private fields that include a progress bar and a slider
	private ProgressBar pb_bar;
	private Slider sli_slider;
```

```
      private VBox vb_mainlayout;
}
```

02) replace the comment for 02 code with the following code

```
      // create all widgets and add them to the layout
      vb_mainlayout = new VBox();
      pb_bar = new ProgressBar(0.5);
      sli_slider = new Slider(0, 100, 50);
      vb_mainlayout.getChildren().addAll(pb_bar, sli_slider);
```

Explanation:

- Although the progress bar represents a range from 0 to 100% internally it is represented as a double from 0 to 1. Thus creating a progress bar with an initial value of 0.5 means that we are defaulting it to 50% to begin with.

03) replace the comment for 03 code with the following code

```
      // set all the ticks for the slider
      sli_slider.setShowTickMarks(true);
      sli_slider.setShowTickLabels(true);
      sli_slider.setMajorTickUnit(10);

      // set the progress bar to grow
      pb_bar.setMaxWidth(Double.MAX_VALUE);
```

Explanation:

- Here note that the maximum width of the progress bar is set to the highest value that a double can take on. The reason we do this is because progress bars by default will not expand to take up an additional horizontal space. This is just to ensure that it takes up the same horizontal space as the slider.

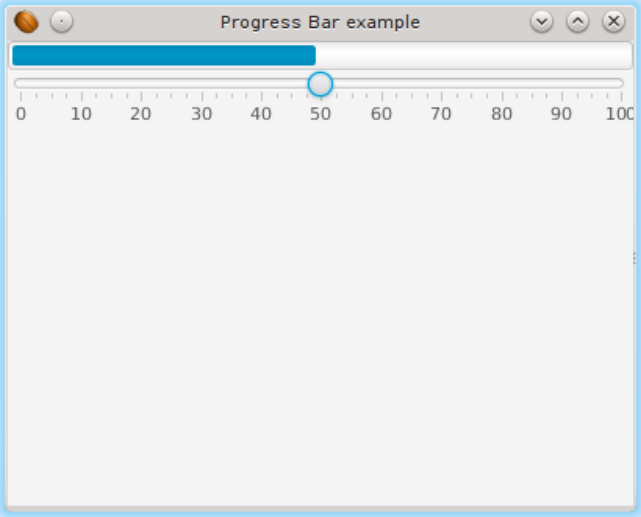
04) replace the comment for 04 code with the following code (good time to compile and run)

```
// set a listener on the slider value property and get it to change the progress
// bar
sli_slider.valueProperty().addListener(new ChangeListener<Number>() {
      // overridden changed method to respond to the change in slider value
      public void changed(final ObservableValue<? extends Number> observable,
final Number oldValue, final Number newValue) {
            // update the value on the progress bar
            pb_bar.setProgress((double) newValue / 100);
      }
});
```

Explanation:

- As before this is setting a change listener on the value property of the slider. Note that when a value is set on the progress bar we have to ensure that the value provided for setting the progress is normalised in the range [0,1].

- running this program will get you the screenshots below which show the slider and progress bar moving in unison.



Example016: The basic ListView and its event handling

In this example the ListView is introduced. It is an extremly powerful widget as it permits you to write your own custom renderer for cells in the list. We will show this in a later example but for now we will show how the basic example of the list view using strings and how to handle the events associated with it.

01) start with a new java project, generate a single source file called Example016.java and give it the following shell code.

```
// simple example that shows the normal list view and how it works with strings

// imports
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example016 extends Application {
	// overridden version of the init method
	@Override
	public void init() {
		// 02 code goes here

		// 03 code goes here

		// 04 code goes here
	}

	// overridden version of the start method
	@Override
	public void start(Stage primaryStage) {
		// set a title, and a scene on the main window
		primaryStage.setTitle("Listview example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden version of the stop method
	@Override
	public void stop() {

	}

	// entry point to our program that will launch our JavaFX application
```

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

	// private fields for this window
	private Label lbl_display;
	private TextField tf_adding;
	private VBox vb_mainlayout;
	private ListView<String> lv_strings;
	private ObservableList<String> ol_strings;
}
```

Explanation:

- As the ListView is adaptable to many different class types it has been made a generic class, thus when you declare a ListView you must also declare the type of object that it will contain and display. Strings are a special case with the ListView. An implementation is already provided for rendering Strings in a list. Therefore if you require a ListView to display strings you do not require extra work.
- Note that there is also an ObservableList that is set to take Strings. The ObservableList contains the data to be displayed in the ListView. Also an ObservableList is one that can signal changes in the data (insert, update, delete, etc) which is perfect for the ListView as any changes in the ObservableList should be reflected in the ListView immediately.

02) replace the comment for 02 code with the following code

```
// initialise the widgets and layouts and add all widgets to the layout
lbl_display = new Label("no item selected");
vb_mainlayout = new VBox();
ol_strings = FXCollections.observableArrayList("one", "two", "three", "four");
lv_strings = new ListView<String>(ol_strings);
tf_adding = new TextField();
vb_mainlayout.getChildren().addAll(lbl_display, lv_strings, tf_adding);
```

Explanation:

- Here we are initialising all of our objects. The first two lines you will have seen many times before. The next line shows how we initialise the ObservableList. Here we use a helper class (FXCollections) to create an ObservableArrayList out of the strings we have provided. Multiple list types are supported but for the sake of conveniece we will use array lists as they are compatible with the ListView.
- The following line will then create the ListView that will render the data inside our ObservableArrayList. As the array list is now connected to the list view any changes in the array list will force an immediate rerender of the list view to reflect the updated data.

03) replace the comment for 03 code with the following code (good place to compile and run)

```
// add in an event handler to the textfield which will add an item into the
```

```
// observable list
tf_adding.setOnAction(new EventHandler<ActionEvent>() {
	// overridden handle method that will add a new item to the list view
	@Override
	public void handle(ActionEvent event) {
		ol_strings.add(tf_adding.getText());
	}
});
```

Explaination:

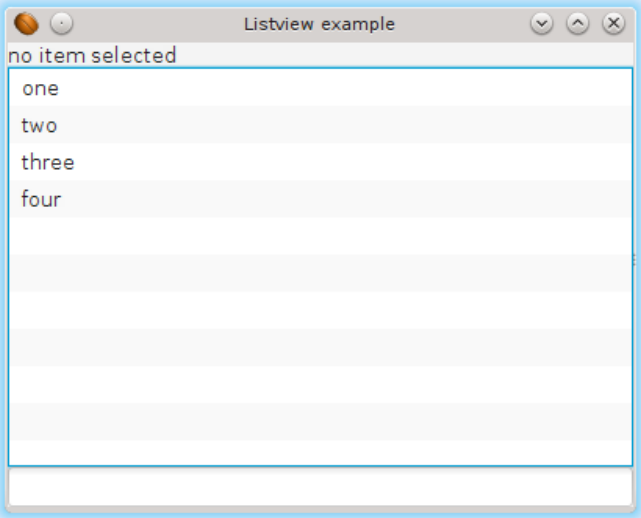
- standard event handler for a text field. Here we are getting the text from the text field and adding it in as a new string into our ObservableList. As soon as the list has been changed you will notice that the ListView updates immediately to reflect these changes.

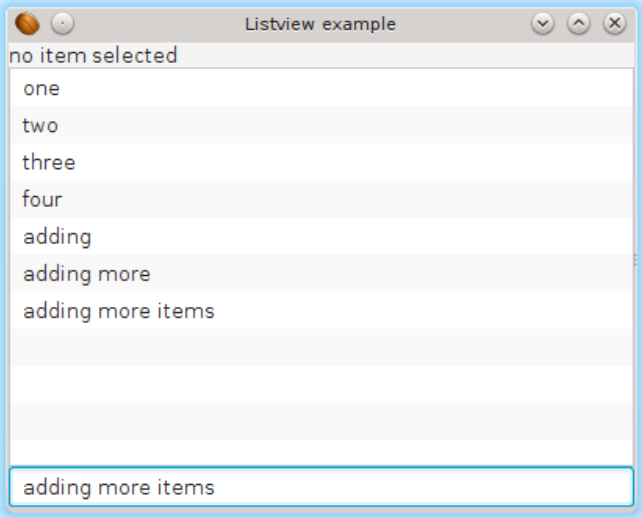
04) replace the comment for 04 code with the following code (good time to compile and run)

```
// add in an event handler to the listview that will listen for events
lv_strings.getSelectionModel().selectedItemProperty().addListener(new
ChangeListener<String>() {
	// overridden changed method that will listen for changes in the selected
	// item. will update the display to reflect this change
	@Override
	public void changed(final ObservableValue<? extends String> observable,
final String old_value, final String new_value) {
		lbl_display.setText("Item selected: " + new_value);
	}
});
```

Explaination:

- Here we are listening for a change in the item that is selected. This listener will be triggered when a user clicks on any of the strings in the list. Note that we have to obtain the selectedItem property of the ListView's SelectionModel in order to add a listener. There is no setOnAction() method so we have to use the addListener() method to add in a new change listener (you've seen these before). Please note that the type of the ChangeListener (the <String> part) must match the type that was set on the ObservableList and the ListView.
- The listener will keep task of the previous item and the new item that was selected. In this case we simply state which item was selected in the list.
- If you run this program now you will obtain the screenshots that you see below. The screenshot on the left shows the program in its initial state. While on the right we have added strings to the list and also have selected an item.


Listview example
no item selected
one
two
three
four


Listview example
no item selected
one
two
three
four
adding
adding more
adding more items
adding more items

Example017: The ListView with a custom cell renderer

In the previous example we introduced the ListView and stated that it is one of the more powerful widgets that JavaFX provides. In this example we will showcase that power by implementing a very simple todo list where each row in the list will have a task item that is composed of a CheckBox and a String. We will also show how to react to events from these items. Please note that the example here is a lot more complex than the previous example however, the structure for generating a custom cell renderer for any item type will be very similar.

01) start with a new java project, generate three source files called Example017.java, TaskItem.java and TaskItemCellRenderer.java. Give TaskItem.java the following code

```
// simple class that is publically accessible to demonstrate the
// custom cell renderer

// class definition
class TaskItem {
	// constructor for the class
	public TaskItem(boolean complete, String name) {
		this.complete = complete;
		this.name = name;
	}

	// create a few public fields for simplicity.
	public boolean complete;
	public String name;
}
```

Explaination:

- As our tasks have to fit into an array list we need to generate a simple class for containing information about our tasks. In this case we have a very simple task that has a name and inidicated if it has been completed. We make the fields publically accessible to simplify the example
  - I'm fully aware this is bad object oriented practice and would encourage you not to use this type of object creation in real projects.

02) Add the following shell code to TaskItemCellRenderer.java

```
// class that implements a custom cell renderer for the ListView.
// this class renders TaskItem objects into ListView cells

// imports
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.ListCell;
import javafx.scene.layout.HBox;

//a cell renderer class that will generate a custom set of view for our task
items
```

```
class TaskItemCellRenderer extends ListCell<TaskItem> {
	// default constructor for this class
	public TaskItemCellRenderer(Example017 ref) {
		// call the superclass constructor
		super();
		this.ref = ref;

		// 04 code goes here

		// 12 code goes here
	}

	// method that will update the display of a list cell whenever it needs to
	// be updated
	@Override
	public void updateItem(TaskItem item, boolean empty) {
		super.updateItem(item, empty);

		// 08 code goes here
	}

	// private fields of the class
	private HBox hb_mainlayout;
	private CheckBox cb_done;
	private Label lbl_task_name;

	// reference to the application so we can update the application should
	// a change happen in a list item
	private Example017 ref;
}
```

Explaination:

- Any time you define a custom cell renderer for the ListView it must override the ListCell class. ListCell is a generic type thus you must also specify what data this ListCell represents. In this case we want to render TaskItem objects thus we need to write ListCell<TaskItem>
- Here you may notice that we have defined a constructor that will take in and copy a reference to our main application. The reasons for this will be explained later. However please note that when you define a constructor for a custom cell renderer the first call you **must** make is to call the superclass constuctor. Failing to do so will result in exceptions.
- Every time you define a custom cell renderer you will be required to override the updateItem() method. This method is used to update ListView cells. For efficiency and memory reasons the ListView will reuse cells as often as possible for rendering. Creation of ListView cells requires a lot of work and time.
    - Like the constructor the first call of this method **must** be a call to the superclass version of the updateItem() call.
    - This method takes two arguments. The first is the item that is to be rendered. This must match the type that you specified when you extended the ListCell

class. The first argument is the item that is to be rendered. It is possible for this value to be null as this may be an empty cell being rendered.

- The second arguement indicates if the cell is empty. I've no idea why this redundancy has been introduced but it is there and the JavaFX javadocs recommend that you use it. We will show its use later.

03) add the following shell code to Example017.java

```
// example that shows how to add in custom cells to the list view widget

// imports
import java.util.ArrayList;

import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListCell;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.Callback;

// class definition
public class Example017 extends Application {
	// overridden init method
	@Override
	public void init() {
		// 05 code goes here

		// 06 code goes here

		// 07 code goes here

		// 09 code goes here

		// 10 code goes here
	}

	// 11 code goes here

	// overridden start method
	public void start(Stage primaryStage) {
		// generate a scene and set a title and size on the window
		primaryStage.setTitle("Custom ListView item example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden stop method
```

```
	public void stop() {

	}

	// entry point into our program that will launch our javafx application
	public static void main(String[] args) {
		launch(args);
	}

	// private fields of the class
	private VBox vb_mainlayout;
	private ArrayList<TaskItem> al_taskitems;
	private ObservableList<TaskItem> ol_items;
	private ListView<TaskItem> lv_items;
	private Label lbl_display;
	private TextField tf_adding;

}
```

Explanation:

- The overall shell of our application. Note that we are using the ArrayList type to collect our starting TaskItems before we embed it inside an observable list. You don't need to do this but for clarity I will generate the ArrayList and provide that to the observable list.

04) replace the comment for 04 code in TaskItemCellRender with the following code

```
	// initialise all of our widgets and layouts
	hb_mainlayout = new HBox();
	cb_done = new CheckBox();
	lbl_task_name = new Label();
	hb_mainlayout.getChildren().addAll(cb_done, lbl_task_name);
```

Explanation:

- Note how there is nothing set on the label or the checkbox. The reason for this is that updateItem will set the appropriate properties for these widgets later on. Setting values on them now will be a waste of time as these properties will be reset later.

05) replace the comment for 05 code in Example017 with the following code

```
	// generate a few task items for the arraylist
	al_taskitems = new ArrayList<TaskItem>();
	al_taskitems.add(new TaskItem(false, "task 1"));
	al_taskitems.add(new TaskItem(true, "task 2"));
	al_taskitems.add(new TaskItem(false, "task 3"));
	al_taskitems.add(new TaskItem(true, "task 4"));
```

Explanation:

- Here we generating a few TaskItems for our ArrayList that will show up in our list.

06) replace the comment for 06 code in Example017 with the following code

```
	// initialise all of our widgets
	vb_mainlayout = new VBox();
```

```
        ol_items = FXCollections.observableArrayList(al_taskitems);
        lv_items = new ListView<TaskItem>(ol_items);
        lbl_display = new Label("no item selected");
        tf_adding = new TextField();
        vb_mainlayout.getChildren().addAll(lbl_display, lv_items, tf_adding);
```

Explaination:

- Standard layout code. Note that in this case we supply and ArrayList to the FXCollections.observableArrayList() method. This will convert the ArrayList into an ObservableList. The rest is all standard initialisation and layout code

07) replace the comment for 07 code in Example017 with the following code

```
// set a cell factory on our list view
lv_items.setCellFactory(new Callback<ListView<TaskItem>, ListCell<TaskItem>>() {
        // method we must override to create new cell items
        @Override
        public ListCell<TaskItem> call(ListView<TaskItem> list) {
                TaskItemCellRenderer r = new TaskItemCellRenderer(Example017.this);
                r.setEditable(true);
                return r;
        }
});
```

Explaination:

- This is an extremly important step in setting up a custom cell renderer. A ListView must have a method to call whenever it requires a new ListCell. This is called the cell factory method. Here we are telling the ListView what method to call by using a generic Callback object, that accepts two generic types.
    - The first type is the type of ListView while the second type is the type of ListCell. In the majority case the generic type for the ListView and the ListCell should match.
- Any implementation of a Callback object must override the call() method. The arguement to the function must be the ListView that was specified as the first type of the Callback and the return value must be the ListCell that was specified as the second type of the Callback object.
    - In most cases you will not need to interact with the list directly as all this method will do is generate an empty cell for the ListView. Note that in this case we have initialised the TaskItemCellRenderer
        - you will see in the constructor this code Example017.this The reason we cannot use the this keyword directly is because using this would refer to the Callback object that we have just defined. Example017.this tells the java compiler that we want to reference the class that contains this anonymous class.
    - We also set the item to be editable. If the item is uneditable then selection of custom cells and also checking of checkboxes would not be possible. In general

if you are defining custom cells it is a good idea to set them editable if the user is supposed to change the state of any cell.

- Finally we return the item to the ListView which will insert it directly in the list

08) replace the comment for 08 code in TaskItemCellRenderer with the following code

```
// update what is displayed for this item assuming that we have an item
if(!empty && item != null) {
	cb_done.setSelected(item.complete);
	lbl_task_name.setText(item.name);
	setGraphic(hb_mainlayout);
} else {
	setGraphic(null);
}
```

Explaination:

- this is the code that goes into the updateItem method of the TaskItemCellRenderer. Here we are updating what is displayed on the cell.
- The first check we have to make before proceeding is to determine if we have any data to render. As was stated earlier there are two variables provided for this check and the official documentation recommends that both are checked to be completely sure that we have no data. Which can be seen in the guard for the if statement
  - If we are satisfied that we have data then we must update the display of this cell. In this case we only have a checkbox and a label to update. The appropriate TaskItem was already passed into the updateItem() function for us so we can map the data to the GUI representation easily.
- Note that in both cases there is a call to setGraphic(). setGraphic() takes in a single Node object that is to be rendered in this cell. All Layouts and Controls are subclasses of the Node class so anything can be used here. In this case we have a simple horizontal layout with a checkbox and a label. Thus to render the lot we provide the horizontal layout as the graphic for this ListCell.
  - If you have no data for this cell you are required to call setGraphic() with the value of null to inform the ListCell that nothing is to be rendered here. Failing to do so will result in graphical glitches.

09) replace the comment for 09 code with the following code

```
// add in an event handler to the listview that will listen for events
lv_items.getSelectionModel().selectedItemProperty().addListener(new
ChangeListener<TaskItem>() {
	// overridden changed method that will listen for changes in the selected
	// item. will update the display to reflect this change
	@Override
	public void changed(final ObservableValue<? extends TaskItem> observable,
final TaskItem old_value, final TaskItem new_value) {
		lbl_display.setText("Item selected: " + new_value.name);
	}
});
```

Explanation:

- ChangeListener for the selectedItem property that is very similar to that in the previous example. The only difference here is that the types have changed from String to TaskItem.

10) replace the comment for 10 code with the following code

```
// add in an event handler for the edit text
tf_adding.setOnAction(new EventHandler<ActionEvent>() {
	// overridden handle method
	@Override
	public void handle(ActionEvent event) {
		// update the list with a new item
		TaskItem temp = new TaskItem(false, tf_adding.getText());
		ol_items.add(temp);
	}
});
```

Explanation:

- Like the previous example this take in the text entered in the TextField and will then convert it into a task item which is added to the observable list. Once the item is added the ListView will then rerender immediately.

11) replace the comment for 11 code with the following code

```
	// public method that will notify us if a task item has changed
	public void taskItemChanged(TaskItem item) {
		lbl_display.setText("state changed on item: " + item.name);
	}
```

Explanation:

- This is a simple method that will state that data within a task item has been changed. The reason for this method will become clear in the discussion of 12 below.

12) replace the comment for 12 code with the following code (good time to compile and run)

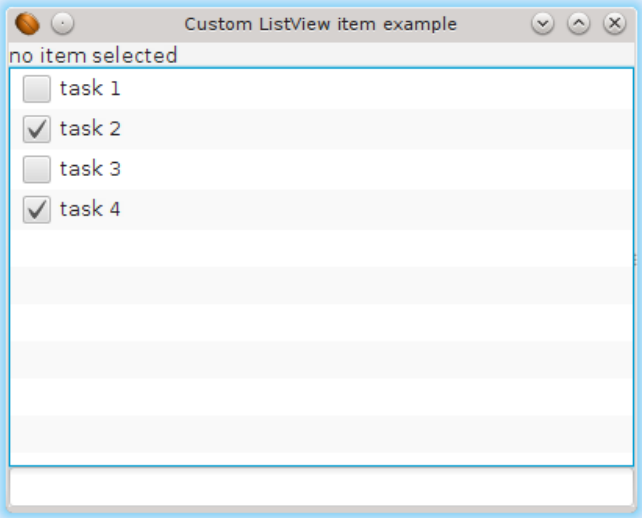
```
// add a listener to the checkbox that will change the state of the item if it
// is clicked
cb_done.setOnAction(new EventHandler<ActionEvent>() {
	// overridden method to handle the change in the checkbox
	@Override
	public void handle(ActionEvent event) {
		// change the state of the item and its complete property
		itemProperty().get().complete = cb_done.isSelected();
		ref.taskItemChanged(itemProperty().get());
	}
});
```

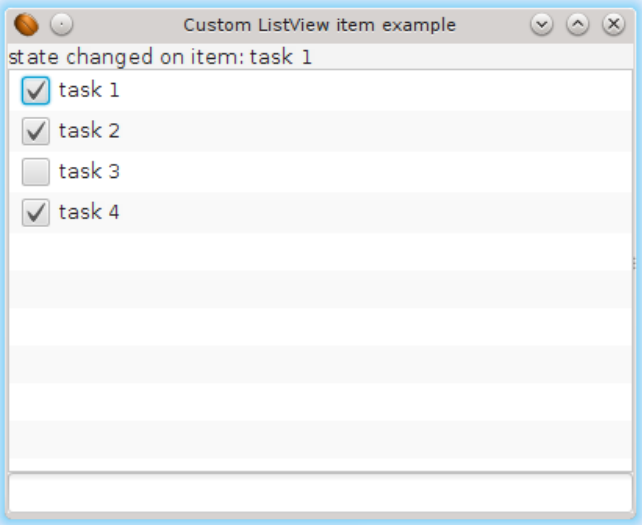
Explanation:

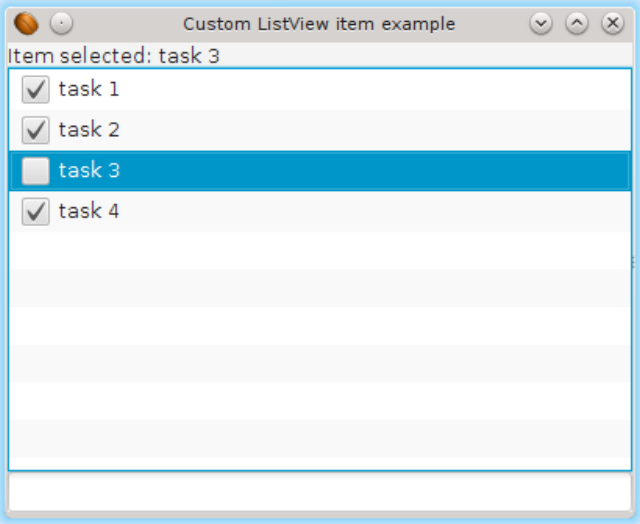
- Standard event handler for a checkbox. The code inside the handle() method is

quite interesting though.

- The first line gets the TaskItem associated with this ListCell (through itemProperty().get()) and then changes the value of the complete boolean to that which is currently set on the check box.
- The second line asks the Example017 object to update itself as there has been a change in the data itself. In this case we tell Example017 which of the TaskItem objects has changed by passing it through this method call. This may seem archaic and a bit long winded, however I've yet to find a clean easy way to register a listener for changes in the state of individual TaskItems that will update the display. Until I find such a method (or the JavaFX developers implement one) this is how I would implement this functionality.
- Running this code will yield the screenshots you see below. The first shows the initial state of the program, the second shows the changing of a task item and the event handling, while the third shows the selection of a task item.







Example018: Introducing the Tab and TabPane.

In this example we will show a common trick that modern webbrowsers and IDEs use to keep multiple documents open at a given time and permit easy switching between them. We will do this through the use of the TabPane and Tab classes. A TabPane may contain zero or more Tab objects. In this example we will use three TextArea objects one for each tab.

01) start a new java project, generate a single source file called Example018.java and give it the following shell code.

```
// simple example that showcases how the tab and tab pane work

// imports
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.control.TextArea;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example018 extends Application {
	// overridden init method
	public void init() {
		// 02 code goes here

		// 03 code goes here

		// 04 code goes here
	}

	// overridden start method
	public void start(Stage primaryStage) {
		// set a scene, title, and size on our window
		primaryStage.setTitle("Tabs example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden stop method
	public void stop() {

	}

	// entry point into our program that will launch our JavaFX application
	public static void main(String[] args) {
		launch(args);
	}

	// private fields of the class
	private VBox vb_mainlayout;
	private TabPane tp_tabs;
	private Tab tab_one, tab_two, tab_three;
```

```
    private VBox vb_one, vb_two, vb_three;
    private TextArea ta_one, ta_two, ta_three;

}
```

02) replace the comment for 02 code with the following code

```
    // initialise our layout and add the tab pane to it
    vb_mainlayout = new VBox();
    tp_tabs = new TabPane();
    vb_mainlayout.getChildren().add(tp_tabs);
```

Explaination:

- Individual Tab objects should never be directly added to a layout. They should be added to a TabPane. Only the TabPane should be added to a layout.

03) replace the comment for 03 code with the following code (good time to compile and run)

```
    // generate a layout for the first tab and add it to the tab pane
    vb_one = new VBox();
    ta_one = new TextArea();
    vb_one.getChildren().add(ta_one);
    tab_one = new Tab("First Area");
    tab_one.setContent(vb_one);
    tp_tabs.getTabs().add(tab_one);
```

Explaination:

- This is the level of work required to initialise and add a Tab to the TabPane. Each Tab will require a layout containing all of the widgets of the tab. In this case it is a single TextArea inside a Vbox. When you create a Tab object it is a good idea to set a title on the Tab (keep it short and informative). To set the layout on the Tab you use the setContent() method.
    - While a layout is recommended the setContent() method will accept any subclass of the Node class.
- Finally to add the Tab to the TabPane we call the getTabs() method of the TabPane and call the add method of that list with our new tab. Once the tab is added the TabPane will rerender its display to show the new tab.

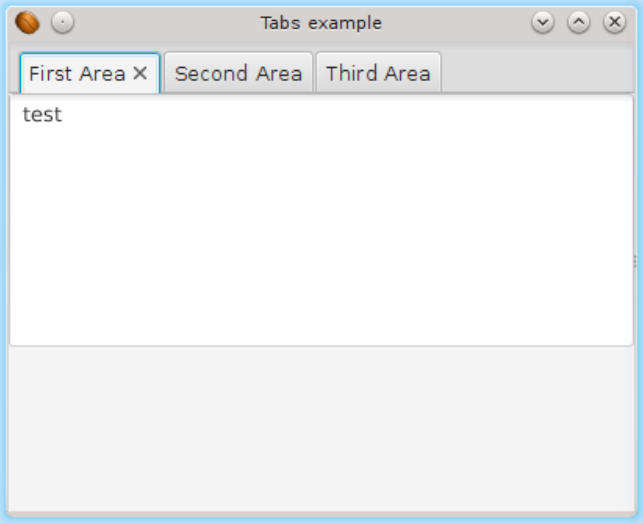
04) replace the comment for 04 code with the following code (good time to compile and run)

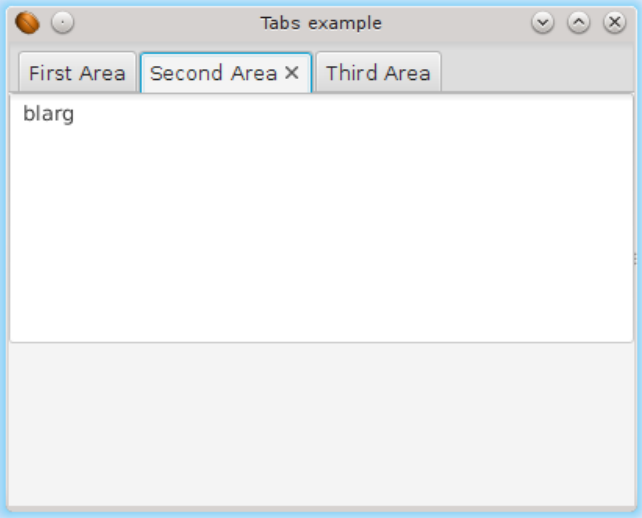
```
    // generate a layout for the second tab and add it to the tab pane
    vb_two = new VBox();
    ta_two = new TextArea();
    vb_two.getChildren().add(ta_two);
    tab_two = new Tab("Second Area");
    tab_two.setContent(vb_two);
    tp_tabs.getTabs().add(tab_two);

    // generate a layout for the third tab and add it to the tab pane
```

```
vb_three = new VBox();
ta_three = new TextArea();
vb_three.getChildren().add(ta_three);
tab_three = new Tab("Third Area");
tab_three.setContent(vb_three);
tp_tabs.getTabs().add(tab_three);
```

Explaination:

- Similar setup for other tabs.
- Running this code will yield you the screenshots below where I have entered text into two different tabs.





Example019: Introducing the DatePicker and its event handling.

In this simple example we will introduce the DatePicker and also how to react to events from the DatePicker when the date is changed.

01) start with a new java project and generate a single source file called Example018.java and give it the following shell code.

```
// simple example that shows how to use the datepicker and handle
// events from it

// imports
import java.time.LocalDate;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// class definition
public class Example019 extends Application {
	// overridden init method
	public void init() {
		// 02 code goes here

		// 03 code goes here
	}

	// overridden start method
	public void start(Stage primaryStage) {
		// put a title, scene, and size on our window
		primaryStage.setTitle("Date Picker example");
		primaryStage.setScene(new Scene(vb_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden stop method
	public void stop() {

	}

	// entry point into our program that will launch our javafx application
	public static void main(String[] args) {
		launch(args);
	}

	// private fields of the class
	private VBox vb_mainlayout;
	private DatePicker dp_date;
	private Label lbl_display;
```

```
}
```

02) replace the comment for 02 code with the following code

```
	// initialise the layout and our widgets
	vb_mainlayout = new VBox();
	dp_date = new DatePicker(LocalDate.now());
	lbl_display = new Label("no date selected");
	vb_mainlayout.getChildren().addAll(lbl_display, dp_date);
```

Explaination:

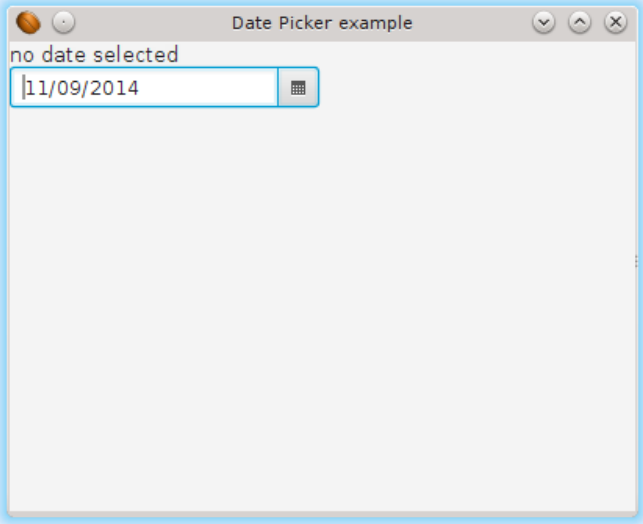
- standard setup code. In this case we initialise the DatePicker with current date by using LocalDate.now() you do not have to provide a starting date for the DatePicker in which case there will be no date displayed.

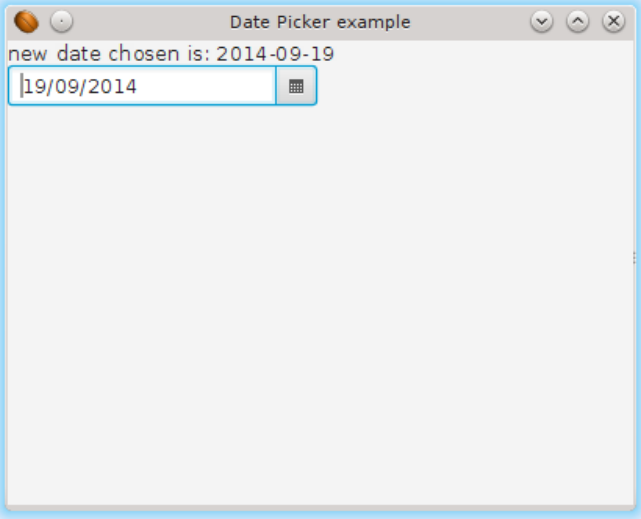
03) replace the comment for 03 code with the following code (good time to compile and run)

```
// set a listener on the date picker so we can react to changes
dp_date.setOnAction(new EventHandler<ActionEvent>() {
	// overridden handle method
	public void handle(ActionEvent event) {
		// set the new date in the label
		lbl_display.setText("new date chosen is: " + dp_date.getValue());
	}
});
```

Explaination:

- Setting the event handler on the DatePicker. The datepicker gives normal ActionEvent which indicate that the Date in the DatePicker has been changed. In this case we simply display that date that the user chose. The getValue() method will return an object of type LocalDate which can easily be changed or adapted into a different date.
- Running this code will get you the screenshots below. The one on the left shows the initial state and the one on the right shows a date being changed.


Date Picker example
no date selected
11/09/2014

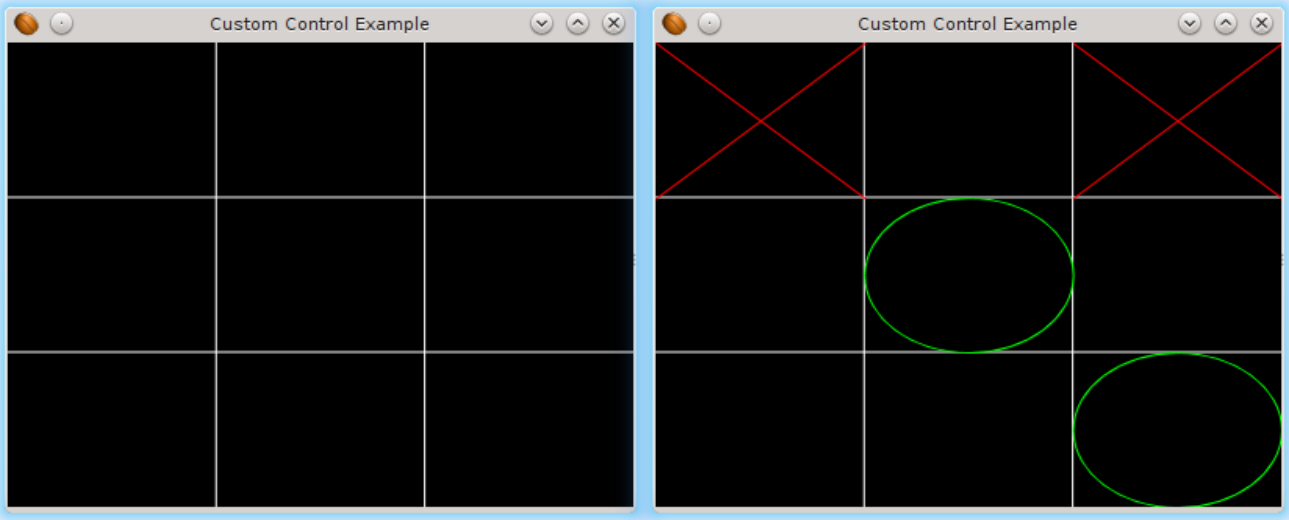

Date Picker example
new date chosen is: 2014-09-19
19/09/2014

Example020: Generating a custom Control and reacting to mouse events and key presses.

Up to this point we have used widgets that are provided by the JavaFX toolkit. However there will be times where you will require a custom widget that is not part of the toolkit. In this example we will show a fully functioning tic-tac-toe (X's and O's or naughts and crosses). There is one caveat though it requires a lot of code compared to other toolkits and this is because we have to work with the JavaFX scene graph and use retained mode rendering instead of the direct drawing commands of immediate mode. The reason we use retained mode rendering is to store as much rendering data in GPU (Graphics Processing Unit) memory which is much quicker than main memory, thus in order to render all that is needed is a single command with the name of the object to render.

Whereas immedate mode rendering will send all data (including the command and data) across the PCI-Express bus immediately this is much much slower than retained mode. Thus the reason we write the extra code is to obtain a performance benefit. There are also other advantages. The JavaFX scene graph may determine that only a small region of the window needs to update. In this case it will only rerender the nodes that have content in that region.

With that in mind I will state that this is an extremly long example but for now I will show you the screenshots of the initial state of the program (left) and a finished game (right) This example shows how to manpulate the scene graph to handle window resizing, placement of objects and reacting to mouse press and key press events.



01) start with a new Java project and generate 5 empty source files called Example020.java, CustomControl.java, CustomControlSkin.java, XOBoard.java, XOPiece.java

02) Add in the following code for Example020.java

```
// JavaFX example that showcases how to build an entirely custom control
// that fits into the JavaFX model.

// imports
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

// class definition
public class Example020 extends Application {
	// overridden init method
	@Override
	public void init() {
		// initialise the stack pane and add a custom control to it
		sp_mainlayout = new StackPane();
		cc_custom = new CustomControl();
		sp_mainlayout.getChildren().add(cc_custom);
	}

	// overridden start method
	public void start(Stage primaryStage) {
		// set a size, title and a scene on the main window. show it when
		// ready
		primaryStage.setTitle("Custom Control Example");
		primaryStage.setScene(new Scene(sp_mainlayout, 400, 300));
		primaryStage.show();
	}

	// overridden stop method
	public void stop() {

	}

	// entry point into our program to launch our JavaFX application
	public static void main(String[] args) {
		launch(args);
	}

	// private fields for this class
	private StackPane sp_mainlayout;
	private CustomControl cc_custom;

}
```

Explaination:

- This is fairly standard code that you have seen thus far. The only major changes we have made here is to take our CustomControl class (defined in the next step) and used it as the control that will be displayed on the stack pane.

03) Add in the following shell code for CustomControl.java

```
// implementation of a custom control for javafx

// imports for the class
import javafx.event.EventHandler;
import javafx.scene.control.Control;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.input.MouseEvent;

// class definition for a custom control
class CustomControl extends Control {
	// constructor for the class
	public CustomControl() {

	}

	// override the resize method
	@Override
	public void resize(double width, double height) {

	}

	// private fields of the class
	private XOBoard xoboard;

}
```

Explaination:

- This is the basic shell for a control. All custom controls must extend the Control class.
- Because we will not be using a skin to display the contents of the control (see next step for reasons why) we will need to override the resize method in case the control is resized. This will permit our control to adapt itself should it recieve extra space or should it be asked to reduce its size.

04) Add in the following shell code for CustomControlSkin.java

```
// basic implementation of a skin

// imports necessary for this class to work
import javafx.scene.control.Skin;
import javafx.scene.control.SkinBase;

// class definition for a custom skin
class CustomControlSkin extends SkinBase<CustomControl> implements
Skin<CustomControl> {
	public CustomControlSkin(CustomControl cc) {
		// call the super class constructor
		super(cc);
	}
}
```

Explaination:

- Ideally in JavaFX a Control would be broken into two pieces. A Control and a Skin. The Control class should implement the logic of the control whereas the Skin should be responsible for rendering it.
- Unfortunately the JavaFX documentation gives very little definitive guidelines for the interaction and design of a skin and a control. Because there is very little documentation we will implement all rendering in the control by manipulating the scene graph. Should better more concrete documentation arrive in future I will rewrite this example to break the control into a control and a skin.
- For now the class you see here is a boilerplate dummy class. The format of the boilerplate is required and should not be changed. If you are making a skin for your own custom control please remember to replace "CustomControl" in this boilerplate by the name of your custom control class

05) Add in the following shell code for XOBoard.java

```
// an implementation of the XO board and the game logic

// imports necessary for this class
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Translate;

// class definition for drawing a game board
class XOBoard extends Pane {
	// constructor for the class
	public XOBoard() {

	}

	// we have to override resizing behaviour to make our view appear properly
	@Override
	public void resize(double width, double height) {
		// call the superclass method first
		super.resize(width, height);
	}

	// public method for resetting the game
	public void resetGame() {

	}

	// public method that tries to place a piece
	public void placePiece(final double x, final double y) {

	}

	// private fields of the class
	private int[][] board;		// array that holds all pieces
	private XOPiece[][] renders;  // array that holds all the render pieces
```

```
        private Rectangle back;       // background of the board
        private Line h1, h2, v1, v2;  // horizontal and vertical grid lines
        private double cell_width, cell_height;   // width and height  of a cell

        // translation of {one, two} cell {width, height}
        private Translate ch_one, ch_two, cw_one, cw_two;
        private int current_player;         // who is the current player

        // constants for the class
        private final int EMPTY = 0;
        private final int XPIECE = 1;
        private final int OPIECE = 2;
}
```

Explaination:

- Here we are defining the XOBoard. This class extends the Pane class. We use the Pane as a blank canvas to organise and display our rendering nodes. Note that there is an overridden resize method (in case the size of the control changes) note that the first line in the overridden method is a call to the super class method. It is recommended that you do this as part of your resize method to make sure the control will resize properly.
- In the private section you will notice a number of fields and different object types. We will describe the purpose of these fields
  - EMPTY, XPIECE, OPIECE: constants that represent an empty cell, a cell containing an X, or a cell containing an O repectively.
  - board: a 3x3 array of cells that will maintain the state of the game.
  - renders: a 3x3 array of XOPiece nodes that are used to render the pieces on the game board. This array maps directly to the board array.
  - back: A rectangle that will be used to draw the black background of the board.
  - h1, h2, v1, v2: horizontal and vertical white lines that will make the grid of our board.
  - ch_one, ch_two, cw_one_cw_two: translate objects that will determine the position of our lines. The idea here is that to reposition the lines we will only alter the x or y (one or two pieces of data) value of the translate instead of setting the x and y of both endpoints of a line (four pieces of data). The less data you transfer to the GPU, the faster your rendering will be.
  - current_player: used to keep track of which player is about to place a piece.

06) Add in the following shell code for XOPiece.java

```
// implementation of an XO Piece

// imports required for this class
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
```

```
import javafx.scene.shape.Line;
import javafx.scene.transform.Translate;

// class definition for an X  or O piece
class XOPiece extends Group {
	// constructor for the class
	public XOPiece(int type) {

	}

	// overridden version of the resize method
	@Override
	public void resize(double width, double height) {
		// call the super class method
		super.resize(width, height);
	}

	// overridden version of the relocate method
	@Override
	public void relocate(double x, double y) {

	}

	// private fields of the class
	private Line l1, l2;	// lines for drawing the X piece
	private Ellipse e;		// ellipse for rendering the O piece
	private int type;	// maintain a copy of the piece type we have
	private Translate pos;  // translate that set the position of this piece
}
```

Explaination:

- The shell of code for rendering a single piece on the board. Note that here we are extending the Group class and not the Pane. The reason for this is that a Group specifies a collection of render objects to be rendered however there will not be anything rendered for the Group itself.
    - Particularly useful as we will not be drawing all pixels in the cell meaning that our background will still remain black.
- Again note the resize method and the call to the superclass verison
- The relocate method is there to reposition this group of render commands in the window. Here will will only modify the pos object (again smaller data transfer == quicker rendering)
- Note that the constructor takes in a type. We will use this to determine if we have an X piece or an O piece. Based on this value we will either render two lines (for X) or a single Ellipse (for O)

07) In the CustomControl constructor add in the following code for initialising and adding in the XOBoard to the control

```
// set a default skin and generate a game board
setSkin(new CustomControlSkin(this));
xoboard = new XOBoard();
getChildren().add(xoboard);
```

Explaination:

- All custom controls as part of initialisation must have a skin set on them. The first line generates a new CustomControlSkin object and links it with the CustomControl.
- After this an XOBoard Pane is generated and add to the render children of this custom control

08) In the XOBoard constructor add in the following code for initialising the arrays

```
// initialise the boards
board = new int[3][3];
renders = new XOPiece[3][3];
for(int i = 0; i < 3; i++)
      for(int j = 0; j < 3; j++) {
            board[i][j] = EMPTY;
            renders[i][j] = null;
      }
current_player = XPIECE;
```

Explaination:

- Here we default the board to have empty pieces. We all default the renders to null such that we do not render any pieces on the board.

09) In the XOBoard constructor add in the following code to initialise the rectangles and lines

```
// initialise the rectangle and lines
back = new Rectangle();
back.setFill(Color.BLACK);
h1 = new Line(); h2 = new Line();
v1 = new Line(); v2 = new Line();
h1.setStroke(Color.WHITE); h2.setStroke(Color.WHITE);
v1.setStroke(Color.WHITE); v2.setStroke(Color.WHITE);
```

Explaination:

- Here we initialise the rectangle and the lines. Note that we do not set dimensions on either as these will be reset as all nodes when shown for the first time will have their resize method called. The only information we will set here is the colours of the background and the lines. As this information will not change we will set it once and forget it.
- setStroke() determines the colour for drawing the edge of a shape
- setFill() determines the colour for drawing the interior of a shape.

10) In the XOBoard constructor add in the following code to initialise the horizontal and veritcal line endpoints

```
// the horizontal lines only need the endx value modified the rest of
// the values can be zero
h1.setStartX(0); h1.setStartY(0); h1.setEndY(0);
h2.setStartX(0); h2.setStartY(0); h2.setEndY(0);

// the vertical lines only need the endy value modified the rest of the
// values can be zero
v1.setStartX(0); v1.setStartY(0); v1.setEndX(0);
v2.setStartX(0); v2.setStartY(0); v2.setEndX(0);
```

Explanation:

- This is the reason why we use the translates to move the lines instead of resetting the endpoints. By doing so we eliminate the need to send 12 coordinates to the GPU every time the window needs to be resized. All lines listed here have their starting coordinates set to the origin (0,0) such that when the lines are translated they will appear at the correct point.
- In the case of the horizontal lines the ending Y value will not change as the line is horizontal. So this value can be fixed here.
- In the case of the vertical lines the ending X value will not change as the line is vertical. So this value can be fixed here.

11) In the XOBoard constructor add in the following code to initialise the translates for the horizontal and vertical lines.

```
// setup the translation of one cell height and two cell heights
ch_one = new Translate(0, 0);
ch_two = new Translate(0, 0);
h1.getTransforms().add(ch_one);
h2.getTransforms().add(ch_two);

// setup the translation of one cell width and two cell widths
cw_one = new Translate(0, 0);
cw_two = new Translate(0, 0);
v1.getTransforms().add(cw_one);
v2.getTransforms().add(cw_two);
```

Explaination:

- Here all translates are initialised to an (x,y) of (0,0) because there these values will need to be adjusted after the first resize. In the case of horizontal lines the only value of the translate that will be modified is the y value and for the vertical lines the only value of the translate that will be modified is the x value.
- In order to use a translate on a line we have to add it to the list of transforms for that line. That list is obtained by calling getTransforms() and the addition is done by calling the add() method
    - Note that the term transform is used here. This is to cover translate, scale, and

rotate operations. For the sake of this example we will stick to translates.

12) In the XOBoard constructor add in the following code to add the components to the board itself

```
// add the rectangles and lines to this group
getChildren().addAll(back, h1, h2, v1, v2);
```

13) In the CustomControl resize method add the following code

```
// update the size of the rectangle
super.resize(width, height);
xoboard.resize(width, height);
```

Explaination:

- As before we call the superclass method to make sure the full resize works.
- Because the control has resized it may have more space avaialble or less space available. To get the XOBoard to resize to the new space we will call it's resize method with the new dimensions.

14) in the XOBoard resize method add the following code to recalculate the width and height of a cell.

```
// figure out the width and height of a cell
cell_width = width / 3.0;
cell_height = height / 3.0;
```

Explaination:

- Here we determine what the width and height of an indiviual cell is. These two pieces of information will enable us to determine which cell the user has clicked on and also to determine where the grid lines and the pieces should be placed.

15) in the XOBoard resize method add the following code to resize the background rectangle and the sizes of the grid lines (good place to compile and run)

```
// resize the rectangle to take the full size of the widget
back.setWidth(width); back.setHeight(height);

// set a new y on the horizontal lines and translate them into place
ch_one.setY(cell_height); ch_two.setY(2 * cell_height);
h1.setEndX(width); h2.setEndX(width);

// set a new x on the vertical lines and translate them into place
cw_one.setX(cell_width); cw_two.setX(2 * cell_width);
v1.setEndY(height); v2.setEndY(height);
```

Explaination:

- The first line sets the background rectangle to take the full width and height of the control and leave no gaps.
- The next block of code positions the first horizontial grid line to be at 1/3rd of the height of the window and the second horizontal grid line to be at 2/3rds of the

window height. While the lines are resized to take up the full width of the window.

- A similar operation is performed for the vertical lines.

16) in the XOBoard placePiece method add the following code to calculate the indicies that a player is trying to put a piece in

```
// translate the x, y coordinates into cell indexes
int indexx = (int) (x / cell_width);
int indexy = (int) (y / cell_height);
```

Explaination:

- here we take the x coordinate of the click and the y coordinate of the click and map to our game board. The reason we do this is that the value of x has a range of [0,width) and y has a range of [0,height). Both of these need to be converted into the range [0,3). This is exactly why we defined the cell width and the cell height to be 1/3 such that when we divide the width and height we will get one of three values 0,1, and 2 which are the indices into our arrays.
- Note the type casting x and y are doubles and these cannot be used as indicies into an array we must convert them to ints instead.

17) in the CustomControl constructor add the following code to listen for mouse click events

```
// add a mouse clicked listener that will try to place a piece
setOnMouseClicked(new EventHandler<MouseEvent>() {
	// overridden handle method
	@Override
	public void handle(MouseEvent event) {
		xoboard.placePiece(event.getX(), event.getY());
	}
});
```

Explaination

- Similar event handler to what you've seen before except instead of using an ActionEvent we use a MouseEvent which contains a lot of information about the mouse. In this case we are only interested in the coordinates of where the user has clicked on the board. We use these values to try and place a piece on the board.

18) in the XOPiece constructor add the following code to initialise the translation of the piece and the type of the piece

```
// create a new translate object and take a copy of the type
pos = new Translate();
this.type = type;
```

Explaination:

- All XOPieces will need to be translated into the correct spot on the control thus we need a Translate object. The type has to be kept as well as this will affect the rendering of the piece itself.

19) in the XOPiece constructor add in the following code to initialise for an X piece

```
// choose which piece type we have
if(type == 1) {
	// we have an X piece generate two lines and add them to
	// as render nodes add in the translate for our lines
	l1 = new Line(); l2 = new Line();
	getChildren().addAll(l1, l2);
	l1.getTransforms().add(pos);
	l2.getTransforms().add(pos);
	l1.setStroke(Color.RED);
	l2.setStroke(Color.RED);

	// as l1 starts top left startx, starty will always be zero
	// as l2 starts top right starty, endx will always be zero
	l1.setStartX(0); l1.setStartY(0);
	l2.setStartY(0); l2.setEndX(0);
}
```

Explaination:

- This is how we initialise an X piece. We generate two red lines add them to the render group with getChildren().addAll(). Note that both lines are sharing the same transform object
- The first line will go from the top left of the cell (0, 0) to the bottom right of the cell (cell_width, cell_height) thus the start x and start y can have fixed values
- The second line will go from the top right of the cell (cell_width, 0) to the bottom left of the cell (0, cell_height) thus the start y and end x can have fixed values.

20) in the XOPiece constructor add in the following code to initialise for an O piece

```
else {
	// we have an O piece generate an oval and add it as a
	// render node
	e = new Ellipse();
	getChildren().addAll(e);
	e.getTransforms().add(pos);
	e.setStroke(Color.LIME);
}
```

Explaination:

- Here we generate an ellipse (oval) for the O piece. As we have not forced the board to be square we must use an oval to fill the full rectangle of the cell.
- Like before we set the transform on the ellipse and set the ellipse to render a green colour.

21) in the XOPiece relocate method add in the following code

```
// call the superclass method and update the position
super.relocate(x, y);
pos.setX(x); pos.setY(y);
```

Explanation:

- used to relocate the cell in the event of the piece being created for the first time or in case of a window resize. All that is done here is to update the position of the translate object and to call the superclass version of the same method.

22) in the XOPiece resize method add in the following code

```
// update depending on the type
if(type == 1) {
      // resize the lines
      l1.setEndX(width); l1.setEndY(height);
      l2.setStartX(width); l2.setEndY(height);
} else {
      // recenter the ellipse// and update the radii
      e.setCenterX(width / 2); e.setCenterY(height / 2);
      e.setRadiusX(width / 2); e.setRadiusY(height / 2);
}
```

Explanation:

- Here is where we resize the piece. As was mentioned in step 19 there are only four values that need to be modified to give the lines the correct length. In the case of the first line we need to reset end x and end y and in the other line we must reset start x and end y to make the lines correct again
- in JavaFX an Ellipse can be defined by a center (x, y) position and also an extend in the horizontal direction and the vertical direction. By using the values (width / 2, height / 2) for the center we place the Ellipse in the center of the cell. The Radii are also set to the same values such that the ellipse will take up the full extent of the cell.

23) in the XOBoard placePiece method add in the following code to place an X piece, position it and render it.

```
// if the position is empty then place a piece and swap the players
if(board[indexx][indexy] == EMPTY && current_player == XPIECE) {
      board[indexx][indexy] = XPIECE;
      renders[indexx][indexy] = new XOPiece(XPIECE);
      renders[indexx][indexy].resize(cell_width, cell_height);
      renders[indexx][indexy].relocate(indexx * cell_width, indexy *
cell_height);
      getChildren().add(renders[indexx][indexy]);
      current_player = OPIECE;

}
```

Explanation:

- The first check determines if the current board piece is empty. If not then no further processing will be performed here. Should the spot be empty we will also ask which player it is. In the case of the XPIECE we will generate an XOPiece to represent an X. It is given a size of (cell_width, cell_height) to take up the full dimensions of the cell and is relocated to the cell that the user clicked on by multiplying in the cell width and cell height. After the move is made the new piece is added to the render

list and the turn changes to the O player.

24) In the XOBoard placePiece method add in the following code to place an O piece, position it and render it.

```
else if(board[indexx][indexy] == EMPTY && current_player == OPIECE) {
	board[indexx][indexy] = OPIECE;
	renders[indexx][indexy] = new XOPiece(OPIECE);
	renders[indexx][indexy].resize(cell_width, cell_height);
	renders[indexx][indexy].relocate(indexx * cell_width, indexy *
cell_height);
	getChildren().add(renders[indexx][indexy]);
	current_player = XPIECE;
}
```

Explanation:

- Similar to 23 above.

25) in the XOBoard resize method add in the following code to resize and relocate the pieces upon resize of the window (good place to compile and run)

```
// we need to reset the sizes and positions of all XOPieces that were placed
for(int i = 0; i < 3; i++) {
	for(int j = 0; j < 3; j++) {
		if(board[i][j] != 0) {
			renders[i][j].relocate(i * cell_width, j * cell_height);
			renders[i][j].resize(cell_width, cell_height);
		}
	}
}
```

Explanation:

- In an earlier step we had to resize and relocate the grid lines when the window size changed. This does the same for each of the pieces. By asking them to resize and relocate.

26) in the XOBoard resetGame method add in the following code to reset the game

```
for(int i = 0; i < 3; i++) {
	for(int j = 0; j < 3; j++) {
		board[i][j] = 0;
		getChildren().remove(renders[i][j]);
		renders[i][j] = null;
	}
}
```

Explanation:

- Here we empty all pieces out of the internal board and remove all pieces from the render list to reset the game.

27) in the CustomControl constructor add in the following code to add in a key listener for resetting the game. (good place to compile and run)

```
// add a key listener that will reset the game
```

```
setOnKeyPressed(new EventHandler<KeyEvent>() {
	// overridden handle method
	@Override
	public void handle(KeyEvent event) {
		if(event.getCode() == KeyCode.SPACE)
			xoboard.resetGame();
	}
});
```

Explaination:

- this is an event handler for keys, note that it is very similar to the mouse click listener defined earlier. In this case we use a KeyEvent instead of a MouseEvent. A key event will tell you if a key has been pressed or released and also exactly which key was pressed
- by calling the getCode() method on the event we can query every key on the keyboard. In this case we look for the keycode representing the space bar and if it was pressed then we will reset the game.