Chapter 12, GUIs

John M. Morrison

July 22, 2022

Contents

1 introduction

The acronym GUI stands for *graphical user interface*. This is the means by which you interact with a modern computer. GUIs allow you to use a pointing device such as a mouse, touchpad, or touchscreen and the keyboard to interact with your computer. They allow you to have several processes open at once and for each to occupy a different window on your desktop.

You use a windowing system; each window holds a program and displays what the program running in it is doing. These windows are containers; the windows themselves along with the visible features inside of them are called *widgets*. At the top of each window is a *title bar*, this often holds the name of the app you are using, some buttons that hide, maximize or dispose of the window, and possibly the name of a file you have open.

The main part of the window is called the *content pane*; this is where all of the "good stuff" goes. Within the content pane there may be *controls* that command the application to do things. Examples of these include widgets such as text areas, menus and menu items, scrollbars, buttons, labels, or checkboxes.

If the application has menus, it will have a *menu bar* that holds the menus. It might also have a toolbar, which would hold other controls or a status bar that sends messages to the user.

A widget is a *top-level widget* if it can reside directly on the desktop. A top level widget in `javafx` is called a *Stage*; stages are instances of the class `javafx.stage.Stage`. Stages can exist directly on your computer's desktop. Inside of a `Stage` goes a `Scene`, which determines the appearance of the content pane. Scenes are containers that hold objects called *Nodes*, that constitute the visible part of your application. Every scene has a *root node*. Nodes can be `Parents`, which can contain other nodes. This relation of containment puts a hierarchical tree structure on a scene which we know as its *scene graph*. You can have several scenes which you might switch in and out of a stage.

2 A Minimal GUI Program

Four packages will become important to us as we develop GUI technique.

- `javafx.application` This contains the `Application` class, which will manage the life-cycle of a javaFX app.
- `javafx.stage` This contains top-level windows, including things such as file choosers, pop-up windows, and `Stage`, which is the top-level container window for an application.
- `javafx.scene` This package and its descendants includes a panoply of things we will press into service, These items include buttons, text areas, menus and slider bars. This is the home of many of widgets.
- `javafx.event` This packages hold classes that are useful in responding to such things as keystrokes, mouse clicks, and the selection of menu items. Things in these packages make buttons and other widgets "live."

Let us begin with a little exercise, in which we produce a program that makes a window and puts a button in the window.

```
import javafx.application.Application;
public class SimpleGUI extends Application
{
}
```

Your attempt to run this is rewarded with an error message.

```
SimpleGUI.java:2: error: SimpleGUI is not abstract and does not
override abstract method start(Stage) in Application
public class SimpleGUI extends Application
```

```
       ^
1 error
```

Now look in the javafx API guide in the class `Application`. We notice a couple of things. Firstly, we notice this.

```
public abstract class Application
extends Object
```

The class `Application` is abstract. Secondly, we see that the method `start` is marked `abstract`. We must implement this method or the compiler will not proceed.

We will need to add an import so the import police do not come down on us. We do this so the class `Stage` is visible.

```
import javafx.application.Application;
import javafx.stage.Stage;
public class SimpleGUI extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
    }
}
```

To execute this class we need a main. Here it is.

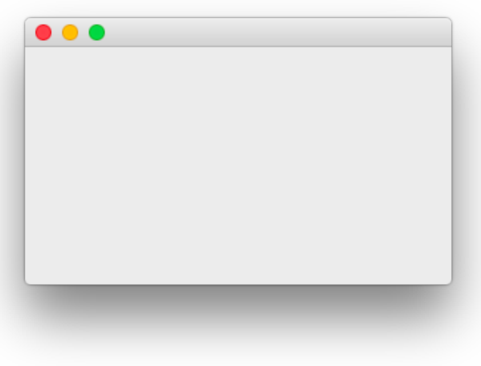
```
import javafx.application.Application;
import javafx.stage.Stage;
public class SimpleGUI extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
    }
    public static void main(String[] args)
    {
    }
}
```

Run this and see nothing. Now for the body of the main. We are calling the static method `launch` in `Application` to launch the application.

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

```
public class SimpleGUI extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

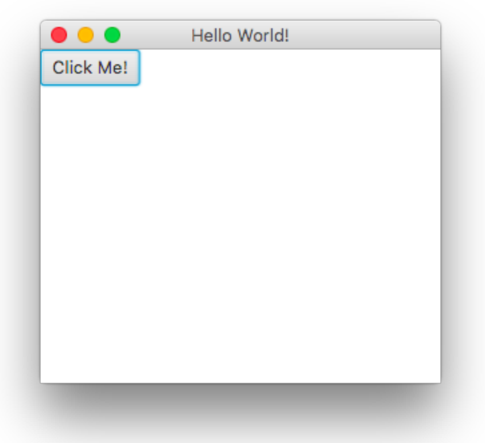
Run this and see an empty window.



Next, we add a button to the window.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.Group;
import javafx.stage.Stage;

public class SimpleGUI extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.setTitle("Hello World!");
        Button aButton = new Button("Click Me!");
        Group root = new Group();
        root.getChildren().addAll(aButton);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Now when you run it, you will see a window pop up on your screen. In the title bar, you will see "Hello World!" The content pane will have a button in it labeled, "Click Me!"



If you are jaded and unimpressed, here is a look at Microsoft Foundation Classes using C++. Feast your eyes below and be appalled at the huge and puzzling program you have to write just to replicate the modest result here we just produced with four lines of code. What's worse is that we don't even get the button!

```
#include <afxwin.h>

class HelloApplication : public CWinApp
{
public:
  virtual BOOL InitInstance();
};
```

```
HelloApplication HelloApp;

class HelloWindow : public CFrameWnd
{
  CButton* m_pHelloButton;
public:
  HelloWindow();
};


BOOL HelloApplication::InitInstance()
{
  m_pMainWnd = new HelloWindow();
  m_pMainWnd->ShowWindow(m_nCmdShow);
  m_pMainWnd->UpdateWindow();
  return TRUE;
}

HelloWindow::HelloWindow()
{
  Create(NULL,
    "Hello World!",
    WS_OVERLAPPEDWINDOW|WS_HSCROLL,
    CRect(0,0,140,80));
  m_pHelloButton = new CButton();
  m_pHelloButton->Create("Hello World!",
      WS_CHILD|WS_VISIBLE,CRect(20,20,120,40),this,1);
}
```

Aren't you glad you saw that?

The `Application` manages the life-cycle of your app. The *primary stage* is the outer skin of the first window of your program. From this primary stage, you can create other stages.

The contents of the stage are kept in a `Scene`. The scene maintains the contents of the content pane. You also see a `Group`; this is the root of the scene graph, which allows us to create a hierarchy of elements placed by us in the scene. Let's look at this segment of code.

```
    Button aButton = new Button("Click Me!");
    Group root = new Group();
    root.getChildren().addAll(aButton);
    primaryStage.setScene(new Scene(root, 300, 250));
```

The first line creates a new button. The second is more mysterious. A group maintains a collection of children that are Nodes that go into the scene graph. So, we first create a group. We then add our button to its list of nodes using the group's `getChildren()` method. This returns an `ObservableList`, which is an interface that extends the familiar `List` interface. The `List` interface has an `addAll` method with a varargs method that allows you to add a comma-separated list of items to it.

We finally set the scene (stuff in the content pane), specifying that the group `root` is to be placed in it and specifying dimensions for the content pane. The elements in the content pane are kept in the scene graph. The scene graph consists of a finite family of rooted trees, one for each scene you are maintaining. One tree from this graph can be placed as the `Scene` in any given `Stage`.

Programming Exercises

1. Add two more buttons to the scene. Give them different labels. Run it and see what it looks like. Use the group's `addAll` method to add all three buttons at once.
2. Change the `Group` to an `HBox`. What does that do? Make sure you do the right import; look in the API guide.
3. Change the `Group` to a `VBox`. What does that do? Make sure you do the right import.

3 The Full Skinny on Application Life Cycle

Here is a general application and exactly what happens when you run it.

```
import import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
public class Example extends Application
{
    public Example()
    {
    }
    @Override
    public void init()
    {
    }
    @Override
    public void start(Stage primary)
    {
    }
    @Override
    public void stop()
    {
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

In general when you create an application, the following occur.

1. You enter `java Example` at the command line.
2. The main method of the application runs, causing the application's static `launch` method to run.
3. The constructor executes.
4. The `init()` method executes. By default this does nothing.
5. The `init()` method returns and then `start()` starts to run.
6. The `start` method is actually a loop. This loop terminates if the user quits the program, if exception or memory error kills it, or if the `start` method returns.
7. If the event that quits the program calls `Platform.exit()` or if the go-away button is clicked, `stop()` is called. You can use this method to save or close opened files, or any other housekeeping that should be done prior to shutting down. The `stop()` method by default does nothing. If the user clicks on the go-away button, the `stop()` method is also called.
8. The `stop()` method returns and the program's process is terminated. The kernel reclaims the program's memory and execution ends.

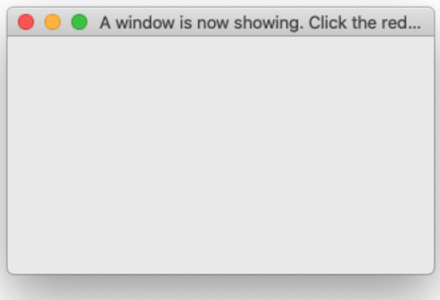
To see all of this, let us put some print statements into our code and run it.

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;
public class Example extends Application
{
    public Example()
    {
        System.out.println("Constructor is constructing.");
    }
    @Override
    public void init()
    {

        System.out.println("init is initting baby booties.");
    }
    @Override
    public void start(Stage primary)
    {

        System.out.println("Start is staring and staying until you quit.");
        primary.setTitle("A window is now showing. Click the red dot to dismiss");
        primary.show();
    }
    @Override
    public void stop()
    {
        System.out.println("Stop is mopping up");
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

If you run this out of a command window, you will see this.

```
unix> javafx Example
Constructor is constructing.
init is initting baby booties.
Start is staring and staying until you quit.
```

You will notice that your command window is now blocked. It is waiting for the child process you spawned to quit. You should see a window that looks like this.



Click its red go-away button and your command window will be relased. It will also tell you that, "Stop is mopping up." You now have visible evidence of the process we just described.

4 Event Handling: buttons

We will begin with the "how" and then get to the "why." Let us make a button, and have it print text to `stdout` when it is clicked.

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.Group;
import javafx.stage.Stage;

public class SimpleGUI extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage)
    {
        primaryStage.setTitle("Hello World!");
        Button aButton = new Button("Click Me!");
        Group root = new Group();
        aButton.setOnAction(e -> System.out.println("I've been clicked"));
        root.getChildren().addAll(aButton);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Let us step through the code. The object `primaryStage` is the captain window of your application. It is possible for it to spawn other stages. You begin by setting the window's title via the call to `setTitle()`. We next create a button named `aButton` with the text `"Click Me!"` emblazoned on it.

Next, we create a root for the first (and only tree) in the scene graph. Following this, we add the button to the scene graph.

A new `Scene` is then installed in the `primaryStage`, and we show it.

No, you are not crazy. We skipped this line.

```
aButton.setOnAction(e -> System.out.println("I've been clicked"));
```

It's the one we do not understand. Let us now go on a treasure hunt and figure out why it works. Begin by going to the API page for `Button` and looking up `setOnAction`. What we see is that this method is inherited from the class `ButtonBase`. Here is its API method detail.

`setOnAction`

`public final void setOnAction(EventHandler<ActionEvent> value)`

Sets the value of the property onAction.

Property description:

```
The button's action, which is invoked whenever the button is fired.
This may be due to the user clicking on the button with the mouse,
or by a touch event, or by a key press, or if the developer programmatically
invokes the fire() method.
```

What we deduce here is that `setOnAction` takes an object of type `EventHandler<ActionEvent>` as its argument. Now it's time to visit this class... er interface. It specifies exactly abstract method, `handle`.

```
@FunctionalInterface
public interface EventHandler<T extends Event>
{
    public void handle(T event);
}
```

What the heck is <T extends Event>?! The `EventHandler` class is a generic class. What you see here is a *type bound*; the type parameter we pass must be subtype of `Event`.

What kind of classes extend `Event`? Let us go to that page. There is a fair-sized list of them, but one of them is `ActionEvent`. This is the type of event that is fired when a button is pushed. If an event handler is attached to the button via `setOnAction`, the code in its `handle` method is executed.

An object of type `EventHandler<ActionEvent>` is an event handler for an action event. It must have one public method,

```
public void handle(ActionEvent event)
{
    //code for the event
}
```

The type parameter is the event type. We have met one event type in creating our live button, `ActionListener`. The button's `setOnAction` method expects an object of type `EventHandler<ActionListener>`. Consider this code fragment.

You see that Java is performing type inference in a manner identical to that of the example we did with the interface `Comparator` in the class `ConcordanceEntry`.

5 Fun with Muhammad Ali: How to Achieve the Desired Layout

The package `javafx.scene.layout` contains classes that control the arrangements of widgets in the content pane or a portion thereof. We have met two already in the exercises, the `VBox` which puts things in a vertical column and `HBox` which arranges them in a horizontal row.

We are now going to meet some of their brethren which will give us a decent palette for creating our own layouts. These classes all descend from `javafx.scene.layout.Pane`. If you go into the JavaFX API guide, you will see these direct descendants of `Pane`. All reside in package `javafx.scene.layout`.

1. AnchorPane
2. BorderPane
3. DialogPane
4. FlowPane
5. HBox
6. PopControl.CSSBridge
7. StackPane
8. TextFlow

9. TilePane
10. VBox

We begin with the border pane. We start by creating a minimal javafx application.

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

public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
        Scene s = new Scene(bp, 500, 500);
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

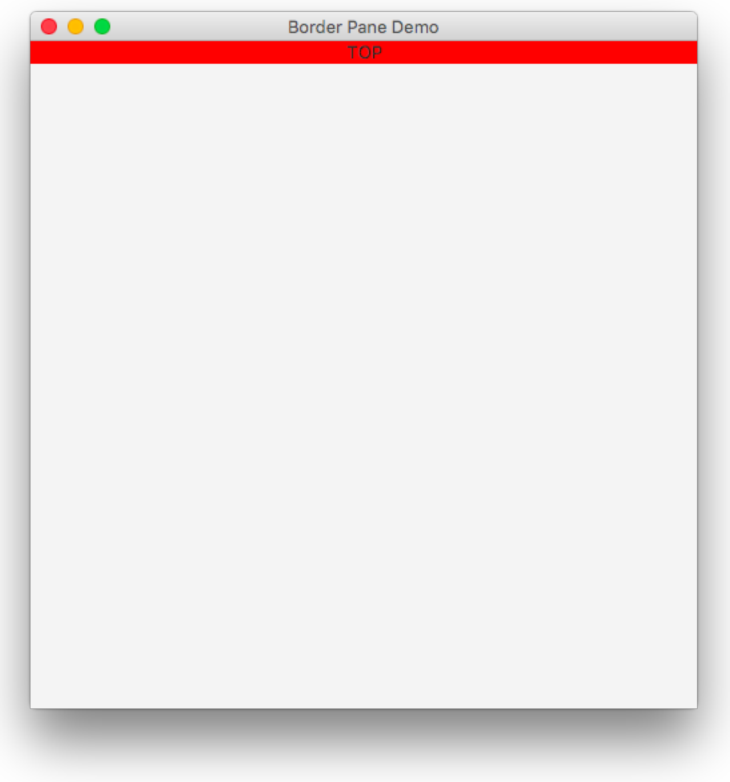
Next, we make a `Scene` with a `BorderPane` it and embed it in our `Stage`.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.BorderPane;

public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
        BorderPane bp = new BorderPane();
        Scene s = new Scene(bp, 500, 500);
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Now we go to work on the top. We make an `HBox` and put a `Label` in it marked `TOP`. We add the label to the `HBox`. We set the `HBox`'s alignment to center, and set its background color to red.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.control.Label;
import javafx.geometry.Pos;

public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
        BorderPane bp = new BorderPane();
        Scene s = new Scene(bp, 500, 500);

        Label topLabel = new Label("TOP");
        HBox topBox = new HBox();
        topBox.setAlignment(Pos.CENTER);
        topBox.getChildren().add(topLabel);
        topBox.setStyle("-fx-background-color:red");
        bp.setTop(topBox);


        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

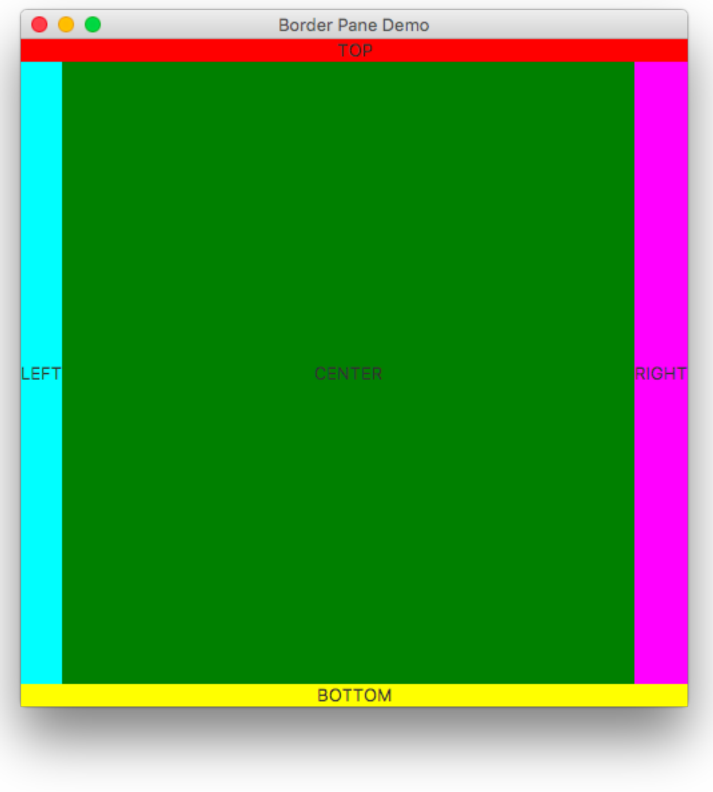
The result can be seen here.



The red field shows the size of the top border pane. Now we add in the rest of the panes, giving them backgrounds of various colors.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.control.Label;
import javafx.geometry.Pos;

public class Border extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
        BorderPane bp = new BorderPane();
        Scene s = new Scene(bp, 500, 500);

        Label topLabel = new Label("TOP");
        HBox topBox = new HBox();
        topBox.setAlignment(Pos.CENTER);
        topBox.getChildren().add(topLabel);
        topBox.setStyle("-fx-background-color:red");
        bp.setTop(topBox);

        Label bottomLabel = new Label("BOTTOM");
        HBox bottomBox = new HBox();
        bottomBox.setAlignment(Pos.CENTER);
        bottomBox.getChildren().add(bottomLabel);
        bottomBox.setStyle("-fx-background-color:yellow");
        bp.setBottom(bottomBox);

        Label leftLabel = new Label("LEFT");
        HBox leftBox = new HBox();
        leftBox.setAlignment(Pos.CENTER);
        leftBox.getChildren().add(leftLabel);
        leftBox.setStyle("-fx-background-color:cyan");
        bp.setLeft(leftBox);

        Label rightLabel = new Label("RIGHT");
        HBox rightBox = new HBox();
        rightBox.setAlignment(Pos.CENTER);
        rightBox.getChildren().add(rightLabel);
        rightBox.setStyle("-fx-background-color:magenta");
        bp.setRight(rightBox);

        Label centerLabel = new Label("CENTER");
        HBox centerBox = new HBox();
        centerBox.setAlignment(Pos.CENTER);
        centerBox.getChildren().add(centerLabel);
        centerBox.setStyle("-fx-background-color:green");
        bp.setCenter(centerBox);

        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Run this and see the result. You now know what a BorderPane looks like.

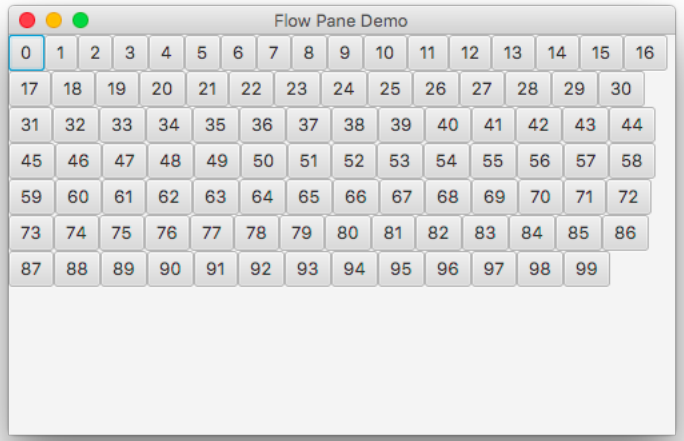


Now let us consider a FlowPane. This has a "jimmybuffetesque" layout policy. Here is a program that puts 100 buttons into FlowPane.

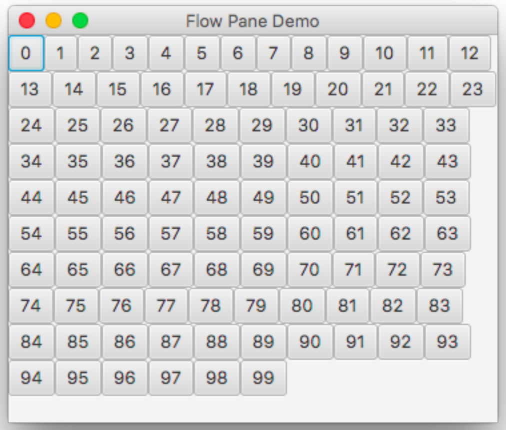
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.Button;
public class Flow extends Application
{
@Override
    public void start(Stage primary)
    {
        primary.setTitle("Border Pane Demo");
                FlowPane fp = new FlowPane();
        Scene s = new Scene(fp, 500, 500);
```

```
                for(int k = 0; k < 100; k++)
                {
                        fp.getChildren().add(new Button("" + k));
                }
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Run the program and you will see this.

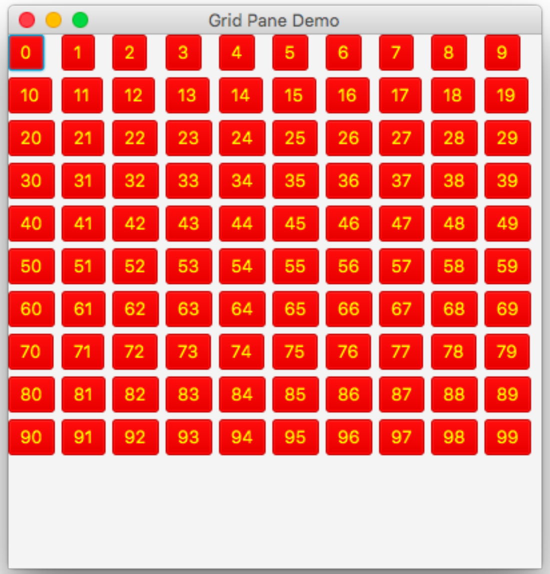


Now resize the window and watch the buttons re-corral themselves amicably. Everybody seems to get along.



Finally, we will do a sample program with a `GridPane`. This will put 100 buttons in a neat little 10×10 grid.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.layout.GridPane;
import javafx.scene.control.Button;
import javafx.scene.paint.Color;
public class Grid extends Application
{
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("Grid Pane Demo");
                GridPane gp = new GridPane();
        gp.setHgap(5);
        gp.setVgap(5);
                for(int k = 0; k < 10; k++)
                {
            for(int l = 0; l < 10; l++)
            {
                Button b = new Button("" + (10 *k + l));
                b.setStyle("-fx-base:red;-fx-text-fill:yellow");
                            gp.add(b, l, k);
            }
                }
        Scene s = new Scene(gp, 400, 400);
        primary.setScene(s);
        primary.show();
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```



A Partial list of Layout Managers	
`Pane`	This just sizes all components to their preferred size. This is the parent class of all of the layout managers.
`StackPane`	This places it children in a back to front stack atop one another.
`FlowPane`	It enforces a "Jimmy Buffet" policy in which widgets go with the flow.
`BorderPane`	This has fields for TOP, LEFT, RIGHT, BOTTOM, and CENTER. The CENTER field is "piggy" and will devour the entire content pane if no other portions of the pane are occupied.
`VBox`	This positions its child widgets vertically.
`BBox`	This positions its child widgets horizontally.
`GridPane`	This positions its child widgets in a rectangular grid.