# CS15 JavaFX Guide

**Table of Contents**

## Application Class/Program Setup

For JavaFX projects, your `App` class (which we will always provide stencil code for) must extend the `javafx.application.Application` class. The `App` class, because it contains the mainline, is the entry point of the program. You must override `Application`'s `start(Stage stage)` method and set up your `Stage`, `Scene` and `PaneOrganizer` classes. Remember, the `PaneOrganizer` class is a convention we use in CS15, so you may see different design patterns in Oracle's official Java tutorials. Regardless, we will hold you to the design standards set in CS15. As a reminder, here is our `App` class from the `ColorChanger` program in the *Graphics Part 1* lecture:

```
public class App extends Application {

      @Override
      public void start(Stage stage) {
            PaneOrganizer organizer = new PaneOrganizer();

            /*
             * The “200” numbers are the width and height of the
             * Scene. Feel free to change them as you wish, and
             * consider making them constants!
             */
            Scene scene = new Scene(organizer.getRoot(), 200, 200);

            stage.setScene(scene);
            stage.setTitle(“Color Changer”);
            stage.show();
      }
}
```

Note: You never need to instantiate a `Stage` manually. A `Stage` is automatically created and passed in when the `start(Stage stage)` method is called.

## Debugging

See our Runtime Exceptions Guide for an explanation of common Java runtime exceptions. The information in the guide is for general Java programs that don't necessarily use JavaFX - while almost all of that information applies for JavaFX programs as well, JavaFX stack traces look somewhat different, so read on.

When you encounter runtime exceptions in your programs, your stack trace might look like this:

```
Exception in Application start method
java.lang.reflect.InvocationTargetException
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.lang.reflect.Method.invoke(Method.java:497)
	at com.sun.javafx.application.LauncherImpl.launchApplicationWithArgs(LauncherImpl.java:389)
	at com.sun.javafx.application.LauncherImpl.launchApplication(LauncherImpl.java:328)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.lang.reflect.Method.invoke(Method.java:497)
	at sun.launcher.LauncherHelper$FXHelper.main(LauncherHelper.java:767)
Caused by: java.lang.RuntimeException: Exception in Application start method
```

```
        at com.sun.javafx.application.LauncherImpl.launchApplication1(LauncherImpl.java:917)
        at com.sun.javafx.application.LauncherImpl.lambda$launchApplication$156(LauncherImpl.java:182)
        at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.NullPointerException
        at IsaacPacman.GameManager.<init>(GameManager.java:71)
        at IsaacPacman.RootManager.<init>(RootManager.java:26)
        at IsaacPacman.App.start(App.java:20)
        at com.sun.javafx.application.LauncherImpl.lambda$launchApplication1$163(LauncherImpl.java:863)
        at com.sun.javafx.application.PlatformImpl.lambda$runAndWait$176(PlatformImpl.java:326)
        at com.sun.javafx.application.PlatformImpl.lambda$null$174(PlatformImpl.java:295)
        at java.security.AccessController.doPrivileged(Native Method)
        at com.sun.javafx.application.PlatformImpl.lambda$runLater$175(PlatformImpl.java:294)
        at com.sun.glass.ui.InvokeLaterDispatcher$Future.run(InvokeLaterDispatcher.java:95)
        at com.sun.glass.ui.gtk.GtkApplication._runLoop(Native Method)
        at com.sun.glass.ui.gtk.GtkApplication.lambda$null$50(GtkApplication.java:139)
        ... 1 more
```

Eeek!!! This stack trace is so long and un-helpful! Where do we even begin?

Let's take a closer look - we see two lines that start with "Caused by." These are the helpful lines! If we look at the second one:

```
Caused by: java.lang.NullPointerException
        at IsaacPacman.GameManager.<init>(GameManager.java:71)
        at IsaacPacman.RootManager.<init>(RootManager.java:26)
        at IsaacPacman.App.start(App.java:20)
```

we see that a `NullPointerException` is being thrown at line 71 of `GameManager.java`. With this information, we can hunt down our bug and squash it!

In general, when a runtime exception is thrown, you will find the helpful "Caused by" information near the very bottom of the stack trace. It might even be a good idea to scroll down to the bottom and work your way up!

## Coordinate System

In the world of JavaFX (and in many other graphics-related contexts), we measure screen coordinates in pixels, starting from the top-left corner of the window, like so:

Note: Negative coordinates are certainly possible! They'll just refer to locations that are outside the bounds of the window.

Sometimes, it may be useful to store an x-y pair of pixel coordinates together. To do this, use `javafx.geometry.Point2D`. To store coordinates, we pass them into `Point2D`'s constructor. For example, if we wanted to store a `Point2D` with the x-y coordinates (345, 600), we would write:

```
Point2D point = new Point2D(345, 600);
```

Then, to access the coordinates, we would call `point.getX()` and `point.getY()`.

Note: `Point2D` can be used for more than pixel coordinates! You can store any arbitrary x-y pair and use it as you wish. The constructor takes in `doubles`, so you can even store decimal values.

## Panes

In addition to `VBox`, `HBox`, and `BorderPane`, which are detailed in the *Graphics Part 1* and *Graphics Part 2* lectures, there are numerous other `javafx.scene.layout.Pane` subclasses. A full list can be found here under the "Direct Known Subclasses" heading. In this guide, we will talk about a few `Pane` types that you can use to make your programs look extra-extra-pretty!

### Backgrounds

To change a Pane's background color, we use the `javafx.scene.layout.Background` and `javafx.scene.layout.BackgroundFill` classes. `BackgroundFill`'s constructor is of the form `BackgroundFill(Paint fill, CornerRadii radii, Insets insets)`. `javafx.scene.paint.Color` is a subclass of `Paint`, so we can pass a `Color` into the constructor (See the Colors section for information on how to create `Colors`). For simple `Pane` backgrounds, we do not need to specify `CornerRadii` and `Insets`. Thus, to create a blue `BackgroundFill`, we would write:

```
BackgroundFill fill = new BackgroundFill(Color.BLUE,
                                 CornerRadii.EMPTY, Insets.EMPTY));
```

Don't worry about understanding `CornerRadii` and `Insets` precisely - our example will create a background with no border.
Now, we create a `Background` using our `BackgroundFill`. We assume we have a `Pane` reference called `pane`, and we set `pane`'s background:

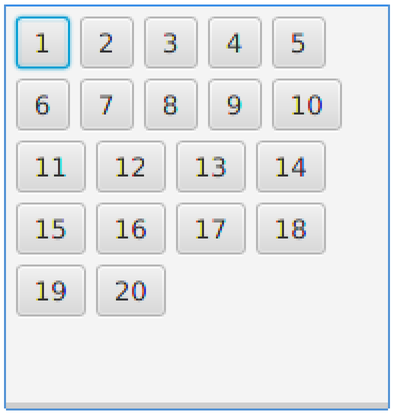
```
Background background = new Background(fill);
pane.setBackground(background);
```

## FlowPane

FlowPane arranges its child nodes in a straight line, one after the other. The layout is horizontal by default, but can be specified as vertical by calling `pane.setOrientation(Orientation.VERTICAL)` . When the "line" of `Nodes` reaches the end of the pane, it wraps around to the beginning and forms a new row or column, depending on whether the orientation is horizontal or vertical. Here, we set up a sample horizontal `FlowPane`:

```
FlowPane flowPane = new FlowPane();
flowPane.setOrientation(Orientation.HORIZONTAL);  // This is the
                                                  // default,
                                                  // so this call
                                                  // is not strictly
                                                  // necessary
flowPane.getChildren().add(new Button("1"));
flowPane.getChildren().add(new Button("2"));
/* Other buttons elided */
```

We see that the rows of buttons wrap around as they reach the right side of the `FlowPane`.

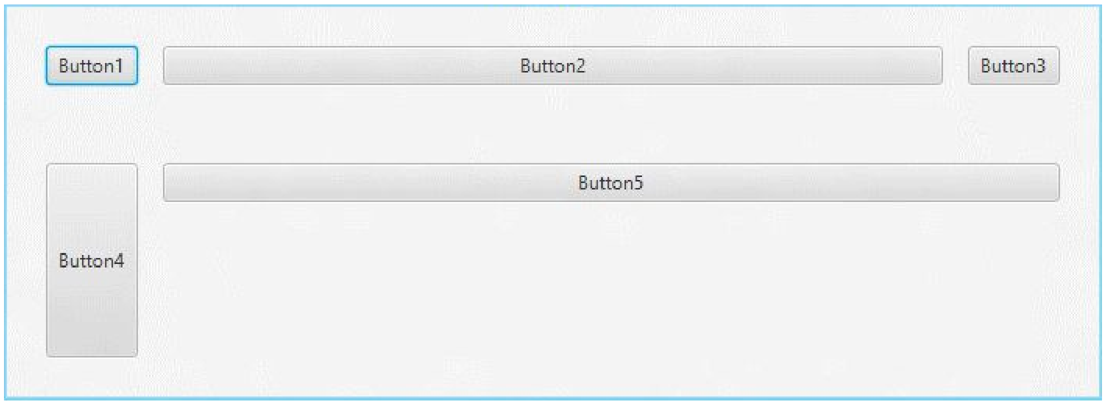


## AnchorPane

AnchorPane allows the positioning of its child `Nodes` relative to the edges of the `AnchorPane`. We simply specify the pixel distance (the "anchor") from the edge of the pane to each `Node` using static methods of `AnchorPane`. If opposite anchors are specified (i.e. left-right or top-bottom) the `Node` will resize itself in that dimension to fulfill both anchor specifications. If only one anchor of a left-right or top-bottom pair is specified, the `Node` will maintain its default or previously specified size in that dimension. Here, we create an example `AnchorPane`:

```
AnchorPane anchorPane = new AnchorPane();
```

```
// Button 1 will “float” in the top-left corner
Button b1 = new Button(“Button 1”);
AnchorPane.setTopAnchor(b1, 10.0);
AnchorPane.setLeftAnchor(b1, 10.0);

// Button 2 will stretch itself left-to-right
Button b2 = new Button(“Button 2”);
AnchorPane.setTopAnchor(b2, 10.0);
AnchorPane.setLeftAnchor(b2, 25.0);
AnchorPane.setRightAnchor(b2, 25.0);

/* Button 3 elided */

// Button 4 will stretch itself vertically
Button b4 = new Button(“Button 4”);
AnchorPane.setTopAnchor(b4, 35.0);
AnchorPane.setBottomAnchor(b4, 10.0);
AnchorPane.setLeftAnchor(b4, 10.0);
AnchorPane.setRightAnchor(b4, 100.0);

/* Button 5 elided */

anchorPane.getChildren.addAll(b1, b2, b3, b4, b5);
```



## GridPane

GridPane allows the layout of child Nodes in a “grid” of rows and columns. The sizes of the rows and columns can be set as pixel measurements or, alternatively, percentages of the GridPane’s size. In addition, each Node’s alignment within its row and column can be set.

GridPane’s rows and columns start from 0, not 1. We do not need to specify the row-column dimensions of the GridPane when we instantiate it - the rows and columns are added automatically as child Nodes are added to the GridPane. To add Nodes to a GridPane, we use the `GridPane`-specific method `add(Node node, int columnIndex, int rowIndex)`. Here, we create a `GridPane` with four columns and one row:

```
GridPane gridPane = new GridPane();

Label label1 = new Label("Andy");
gridPane.add(label1, 0, 0);  // Label added to column 0, row 0

Label label2 = new Label("van");
gridPane.add(label2, 1, 0);  // Label added to column 1, row 0

Label label3 = new Label("Dam");
gridPane.add(label3, 2, 0);  // Label added to column 2, row 0

Button button = new Button("Damn!");
gridPane.add(button, 3, 0);  // Button added to column 3, row 0
```

Next, we set the sizes of the columns to each be 25% of the `GridPane`'s width using the `javafx.scene.layout.ColumnConstraints` class and a "for" loop.

```
for (int i = 0; i < 4; i++) {
      ColumnConstraints cc = new ColumnConstraints();
      cc.setPercentWidth(25);
      gridPane.getColumnConstraints().add(cc);
}
```

The `ColumnConstraints` are applied to the columns in the order they are added. Since we add four `ColumnConstraints`, they are added to the first four columns, which are the columns we are using.

Note: The percentages do not have to all be the same; they are arbitrary. To use multiple percentage values, we would have to create the `ColumnConstraints` individually instead of in a "for" loop.

Additionally, instead of percentages, the `ColumnConstraints`' width can be set to an absolute pixel dimension by passing in the desired number of pixels into the constructor (i.e. `new ColumnConstraints(50)` for a width of 50 pixels).

There is an analogous `javafx.scene.layout.RowConstraints` class for setting the heights of rows. In our example, we have only used one row. We set its height using a pixel dimension:

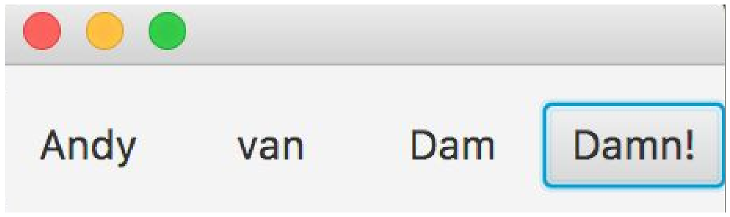
```
RowConstraints rc = new RowConstraints(50);
```

```
gridPane.getRowConstraints().add(rc);
```

Finally, we set the alignment of the child `Nodes` within their "cells" using the static `setHalignment(Node node, HPos value)` and `setValignment(Node node, VPos value)` methods of `GridPane` and a "for-each" loop. There are static `CENTER`, `LEFT`, and `RIGHT` constants for `HPos`, and static `CENTER`, `BOTTOM`, and `TOP` constants for `VPos`:

```
for (Node child : gridPane.getChildren()) {
      GridPane.setHalignment(child, HPos.CENTER);
      GridPane.setValignment(child, VPos.CENTER);
}
```

As with the `ColumnConstraints` percentages, the alignments here do not have to all be the same. To use different alignment values for different children, we would have to set each child's alignments individually instead of in a "for-each" loop.



After all of our work, here is our very pretty result!

## Buttons

Buttons are necessary for almost any User Interface and you're going to be implementing lots of them during your most beautiful and lovely CS15 journey.

There are a few types of buttons and they can have many differences but one thing they all have in common is that they are action producers, meaning they trigger a reaction when they are clicked/selected!

You'll mostly be dealing with two types of buttons in CS15, the conventional button `javafx.scene.control.Button`, and the radio button `javafx.scene.control.RadioButton`.

### Button

Let's go over the normal `Button` first! The role of this button is to produce an action when clicked; very simple. You can instantiate a `Button` using any of the three constructors depending on how many parameters you want to enter. The first constructor will produce a `Button` with an empty text caption:

```
Button button = new Button();
```

The second constructor takes in a string and produces a `Button` with the string entered as its text:

```
Button button = new Button("Example");
```

The third constructor takes in a `String` and an `Image` and produces a button with the `String` entered as its text and and the `Image` entered. You will probably not be using this unless you decide to be creative, which of course is highly encouraged!

For this constructor, you first need to instantiate the `Image`:

```
Image icon = new Image(getClass().getResourceAsStream("icon.png"));
```

then produce the actual `Button`. (note that the above line assumes you have a .png picture of your icon in your project folder.)

```
Button button = new Button("Accept", new ImageView(icon));
```

You can always create an empty `Button` then use the following methods to give it text and/or an icon:

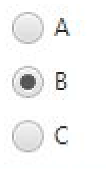
the `setText(String text)` method
the `setGraphic(Node icon)` method

You now know how to make a `Button`, but how do we make it respond to clicks? Simple, use the buttons `setOnAction(EventHandler handler)` which takes in a `javafx.event.EventHandler` as its parameter, and will execute the `handle(ActionEvent event)` method of the handler when clicked! The `handle(ActionEvent event)` method of the handler entered as a parameter will determine what happens when the button is clicked! Note that the parameter `event` of type `ActionEvent` is automatically entered by the button itself and contains information about the `ActionEvent.`
So suppose you have a class `ButtonHandler` that implements `EventHandler` and that already has a defined method `handle(ActionEvent event)` that moves a shape on your screen. If you want to make a button that moves that shape on the screen when clicked, the code would look like this:

```
// instantiating the button with a specific text
Button button = new Button("Move the shape!")
button.setOnAction(new ButtonHandler());
// We are basically telling the button to call the instance of our
// ButtonHandler's handle(ActionEvent e) method when clicked.
```

## Radio Buttons

Now let's talk about `RadioButtons`; these are a little more complicated and confusing than our traditional `Buttons`. They have two states, selected and unselected, and they often appear in groups where only one button can be selected at once.



There are multiple constructors we can use to create instances of this class, but let's focus on two: a normal constructor that creates a `RadioButton` to which you can later assign a `String` using the `setText(String text)` method, and a constructor that initially takes in the `String`.

Here are both constructors:

```
RadioButton rb1 = new RadioButton();
RadioButton rb2 = new RadioButton("example");
```

Your `RadioButtons` will probably be mutually exclusive and the way to do that is put them all in a `javafx.scene.control.ToggleGroup`. This will mean that when one is selected, any already selected `RadioButton` will be deselected, meaning there can only be one selected `RadioButton` in any `ToggleGroup`. Let's create our `ToggleGroup` first:
`ToggleGroup group = new ToggleGroup();`

Now that we have a `ToggleGroup`, we can tell our previously instantiated `RadioButton` to belong to that `ToggleGroup`:

```
rb.setToggleGroup(group);
```

Let's make more `RadioButtons` and add them to our group:

```
RadioButton rb2 = new RadioButton("example2");
RadioButton rb3 = new RadioButton("example3");
rb2.setToggleGroup(group);
rb3.setToggleGroup(group);
```

We can even choose if a `RadioButton` is selected when the program is launched:

```
rb.setSelected(True);
```

You can always know which `RadioButton` in a `ToggleGroup` is selected by calling the `getSelectedToggle()` method on the `ToggleGroup`.

The handling is the same as the normal button. One thing to keep in mind though is that when a `RadioButton` that belongs to a `ToggleGroup` is selected, that means the other `RadioButtons` in the same `ToggleGroup` are deselected so make sure that every `RadioButton` handler negates/ends the actions of the previous `RadioButton` before executing what the selected `RadioButton` is supposed to do.

## Labels

`Label`s are the backbone of any user interface, they tell the user what everything on the screen means. You're going to be implementing a lot of them in CS15, luckily, they are easy to implement. So to create a `javafx.scene.control.Label` use one of the following constructors:

```
Label label = new Label();  //creates an empty label
Label label = new Label("Example!!");  //creates a label with text
```

You can also manipulate the color and font of the `Label` as well as reset the text if you choose too, the following method examples might be useful. For more complex methods, refer to the `Label` Javadocs :

```
label.setText("Any string");  // Sets the labels text to a string

// Sets the label's font to 30pt Arial; note that we create a
// font using the font constructor Font(String font, int size)
label.setFont(new Font("Arial", 30));

// Sets the label's color to the color consisting of the entered
// values of red, green and blue.
label.setTextFill(Color.rgb(20, 20, 20));
```

If your `Label` doesn't fit where you want it to or goes horizontally without going down another line, you might consider wrapping it using the following method:

```
label.setWrapText(true);
```

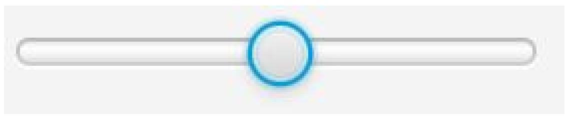
You can even apply really cool effects on `Label` using rotation and translation methods, example:

```
label.setRotate(180);  //rotates the label 180 degrees
//translates the label horizontally 120 pixels
label.setTranslateX(120);
```

## Slider

`javafx.scene.control.Slider` is an effective tool to use in JavaFX applications. It is used to display a continuous or discrete range of valid numeric choices and allows the user to interact with the control. It consists of a track and a thumb. You should be comfortable with implementing them in your CS15 projects!

Example of a `Slider` in Mac OS X:



Let's try making one first! This is the default constructor for a `Slider`:

```
Slider(double min, double max, double value)
```

The constructor takes in the following three parameters:

- `min` - minimum value of the slider, must be double
- `max` - maximum value of the slider, must be double
- `value` - default value of the slider, must be within the range of min to max

```
Example:
// This creates a slider with the minimum value of 0, maximum value
// of 100 and default value of 50
Slider slider = new Slider(0, 100, 50)
```

The `Slider` has many other methods that you could incorporate to customize your `Slider`. Please refer to its Javadocs for more details. One method to keep in mind is the `getValue()` method. It returns the value (as a `double`) that the `Slider` is currently at.

In order to handle a change in the `Slider`, you have to create a class that implements `ChangeListener` (which must implement the method `public void changed(...)` ) You must specify what you what your application to do when the `Slider` is modified in the `public void changed(...)` method.

Then, you **must** remember to add the class that implements `ChangeListener` to the slider's value property. By doing this, the slider's value property will **listen** for **changes,** and it will call your `changed(...)` method every time the slider is modified.

Here's an example in which we create a `Slider` that moves a `Rectangle` from side-to-side: (Focus on how the `SliderListener` is added and implemented; feel free to copy this code to Sublime or Eclipse and run it yourself!)

```
//import statements elided

public class Sample extends Application{

      //main method elided

      public void start(Stage primaryStage) throws Exception{
            Slider slider = new Slider(0, 100, 50);
            Rectangle rec = new Rectangle(50, 50);
            slider.valueProperty().addListener(new
            SliderListener(rec));
            // Set initial x position as the slider's value
            rec.setX(slider.getValue());
            rec.setY(50); // so that the slider and rec don't overlap
            Pane root = new Pane();
            root.getChildren().addAll(slider, rec);
            primaryStage.setScene(new Scene(root, 300, 250));
            primaryStage.show();
      }

      private class SliderListener implements ChangeListener<Number>{
            private Rectangle _rec;

            // Constructor takes in a Rectangle as a parameter so that
            // it can change the Rectangle's position whenever the
            // Slider's value changes
            public SliderListener(Rectangle rec){
                  _rec = rec;
            }

            public void changed(ObservableValue<? extends Number>
observable, Number oldValue, Number newValue) {
                  // changes the rectangle's x position when the slider
                  // is changed
                  _rec.setX(newValue.doubleValue());
            }
      }
}
```

## Colors

*There are three different color classes in Java 1.8 - `java.awt.color`, `com.sun.prism.paint.color` and `javafx.scene.paint.Color`. Make sure that you import the correct one - `javafx.scene.paint.Color`!*

You can find the JavaFX `Color` Javadocs here.

Making `Color` in Java is very simple. There are three ways of doing so but we will only cover two for now. Here is the first way to create a `Color`:

Example:

```
Color c = Color.RED;
```

There are several static variables of the `Color` class, such as `RED, BLUE, WHITE, BLACK,` etc. Check out the Javadocs for more colors!

Second way of creating a `Color` is using the static `rgb(int red, int green, int blue)` method in the `Color` class.

The parameters `red`, `green`, and `blue` correspond to the red component, green component and blue component respectively, and they all have to be `ints` belonging to the range 0 - 255.

Example:

```
// This is pure red, but you get the idea! This has an implicit alpha
// of 1.0. (What is alpha? Read on!)
Color c = Color.rgb(255, 0, 0)
```

Each `Color` also has an alpha value from 0.0 - 1.0 that defines the opacity of the `Color`. An alpha value of 0 is completely transparent, and alpha value of 1 is completely opaque.

```
public static Color rgb (int red, int green, int blue, double
opacity);
```

The above method has an extra parameter of `opacity`, which only accepts values from 0.0 to 1.0.

Example:

```
// This red color is 50% transparent
Color c = Color.rgb(255, 0, 0, 0.5 );
```

If you are picky with your `Colors`, you can consult this Color Picker for specific RGB values of the `Color` you wish to make.

## Shapes

You will make heavy use of `Shapes` this year, as they are needed to visually create some of the most important parts of your projects (e.g. the Doodle in Doodle Jump, and Pieces in Tetris). The JavaFX `Shapes` documentation is more understandable than some other Javadocs (e.g. `Node` Javadocs), so be sure to follow along as this section frequently refers to those Javadocs.

### Types

If you look at the **Direct Known Subclasses** of `Shape`, you can see all of the different types of `Shapes`. You will deal frequently with `Rectangles` and `Ellipses`, and you may also work with `Circles`, `Lines`, `Polygons`, and `Text` at some point during the course.

### Sizing

`Shape` is a subclass of `Node`, but it is a special kind of `Node` in multiple ways. Firstly, unlike `Panes`, `Shapes` do not have children `Nodes`. Thus, you are not able to call `shape.getChildren().add(...)`.

Additionally, `Shapes` are "non-resizable" `Nodes`. Initially, this seems confusing. Can't I change the width and height of my `Rectangle`? What "non-resizable" really means is that `Shapes` are not resized *by their parents*.

The documentation of `Node`'s `isResizable()` method provides a good explanation of this:

> "If this method returns false, then the parent cannot resize it during layout... `Group`, `Text`, and all `Shapes` are not resizable and hence depend on the application to establish their sizing by setting appropriate properties (e.g. width/height for `Rectangle`, text on `Text`, and so on). Non-resizable `Nodes` may still be relocated during layout."

What this means is that a `Rectangle` uses its width and height properties to size itself, rather than constraints determined by its parent. This is different from `Buttons` and `Labels`, which *are* resized by their parents. See the `GridPane` or `AnchorPane` section of this doc for an example of `Labels` and `Buttons` being resized based on constraints.

One aspect of setting `Shape` size that could pose some problems is the fact that size properties aren't part of the `Shape` superclass; rather, different specific size properties exist for each subclass. For example, `Rectangles` have `width` and `height`, whereas `Ellipses` have `radiusX` and `radiusY`. Thus, setting the size of a `Rectangle` would look different than setting the size of an `Ellipse`:

```
Rectangle rectangle = new Rectangle();
rectangle.setHeight(10.0);
rectangle.setWidth(50.0);

Ellipse ellipse = new Ellipse();
ellipse.setRadiusX(40.0);
ellipse.setRadiusY(20.0);
```

This doesn't seem like a *huge* problem, but what if you want to refer to `Shapes` polymorphically? Let's say you have a `List` of `Shapes`, and you want to resize every `Shape` in your list. The following code **wouldn't work** because not all `Shapes` have `setWidth()` and `setHeight()` methods; those are specific to `Rectangle`.

```
List<Shape> shapeList = new ArrayList<>();
/* code that populates list elided */
for (Shape shape : shapeList) {
      shape.setWidth(50.0);
      shape.setHeight(100.0);
}
```

If you do need a way to resize `Shapes` polymorphically, a couple alternatives include using a `Scale` transformation, or using the `Shapes'` `scaleX/Y` properties, which `Shape` inherits from `Node`.

## Setting Location

Setting the on-screen location of `Shape`s is one of the least straight-forward parts of JavaFX. There are many different ways to set a `Shape`'s location, so here is an explanation of each of the different ways to do so. The most important thing to remember is **to choose only one of the following groups of methods when setting Shape location.** For example, don't use both `setLayoutX()` and `setTranslateX()`, otherwise the behavior will not be what you expect or desire.

### Using the `Shape` subclass' methods

**Description:**

Some shape subclasses have their own specific methods for setting location.

**Rectangle:**
`javafx.scene.shape.Rectangle` has `setX(double y)` and `setY(double y)` methods for setting location. These methods set the location of the upper-left corner of the `Rectangle` relative to the upper-left corner of its parent `Node`. If we had a `Rectangle` variable called `rect` that we wanted to move to (42, 85.5), we could write:

```
rect.setX(42);
rect.setY(85.5);
```

**Circle/Ellipse:**
`javafx.scene.shape.Circle` and `javafx.scene.shape.Ellipse` have `setCenterX(double value)` and `setCenterY(double value)` methods for setting location. These methods set the location of the **center** of the shape relative to the upper-left corner of its parent `Node`. If we want to set a `Circle` or `Ellipse`'s location based on the upper-left corner of its bounding box instead, we can write wrapper methods:

```
public void setLocation(Circle circle, double x, double y) {
      circle.setCenterX(x + circle.getRadius());
      circle.setCenterY(y + circle.getRadius());
}

public void setLocation(Ellipse ellipse, double x, double y) {
      ellipse.setCenterX(x + ellipse.getRadiusX());
      ellipse.setCenterY(y + ellipse.getRadiusY());
}
```

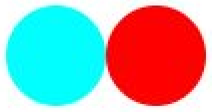
**Polygon/Line:**
`javafx.scene.shape.Polygon` and `javafx.scene.shape.Line` do not have subclass-specific methods for setting location.

**Known Issues:**
`Polygon` and `Line` do not have subclass-specific methods, so for these classes, you'll have to use one of the other ways to set location.

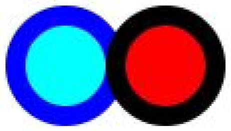
The subclass-specific methods ignore the stroke (border) of the `Shape`, if a stroke exists. By default, a `Shape`'s stroke type is set to `javafx.scene.shape.StrokeType.CENTERED` , which means that half of the stroke width goes on the outside of the actual shape boundary, and half goes on the inside. Because the subclass-specific methods ignore the stroke, the strokes of two `Shapes` will overlap if the `Shapes` are set to adjacent locations. To avoid this, we can set a shape's `StrokeType` to `StrokeType.INSIDE`:

```
shape.setStrokeType(StrokeType.INSIDE);
```

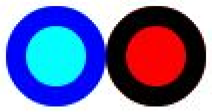
Here, we have two `Circles` with no stroke. They are perfectly adjacent to each other.



Now, without changing the `Circles'` location or size, we add a very wide stroke. We see that the `Circles` now appear to overlap:



If we set the `Circles' StrokeType` to `StrokeType.INSIDE`, the `Circles` no longer appear to overlap:



Because the stroke is now entirely on the inside of the `Circle` boundary, the `Circles` appear smaller. However they are still the same size as the `Circles` without any stroke. The `Circles` with the default `StrokeType` appear larger because half of the stroke is outside of the `Circle` boundary. This is why the `Circles` appear to overlap!

## Using `relocate()`

**Description:**

Each `Shape` inherits this method from the `Node` superclass. It sets the on screen location of the top left corner of the `Shape`'s bounding box to the coordinates passed into the method. A "bounding box" is the unrotated rectangle that surrounds a `Shape` and takes its graphical properties (e.g. rotation, stroke (border), size) into account. Here are some examples of bounding boxes.

Using the `relocate()` method differs from using the other location-setting methods in a couple important ways:

- **It includes the `Stroke` (border) of the `Shape` when positioning the `Shape`**
    - This is because the Shape's stroke is included in its bounding box
- It always sets the position based on the **top left corner** of the `Shape`'s bounding box, whereas the other location-setting methods set the `Shape`'s location based on different parts of the `Shape` depending on the subclass
    - For example, with the other methods, `Rectangles'` locations are set based on their top left corner, but `Ellipses'` locations are set based on their center

`relocate()` adjusts the `layoutX` and `layoutY` properties of a `Shape` in order to set its on screen location. However, it doesn't necessarily directly set `layoutX` and `layoutY` to the passed in `x` and `y`. This is described more in-depth in the Known Issues section.

See the `setLayoutX()` and `setLayoutY()` section for a direct comparison between using `setLayoutX/Y` and `relocate()`.

**Known Issues:**
`relocate()` doesn't necessarily directly set the `layoutX` and `layoutY` properties of the node with the parameters' values, but insteads sets them relative to the `Shape`'s bounding box.

Why should you care about this? This could cause your program to behave unexpectedly if you want to access the location of your `Shape` after having used `relocate()`. Using `getLayoutX()` and `getLayoutY()` to obtain the location of your `Shape` will not necessarily return the same `x` and `y` values passed into `relocate(x, y)`. This is because the `layoutX` and `layoutY` properties do not represent the top left corner of the `Shape`'s bounding box, which `relocate()`'s functionality is based on.

For a detailed example of this with diagrams, see the `setLayoutX()` and `setLayoutY()` section.

## Using `setLayoutX()` and `setLayoutY()`

**Description:**
Each `Shape` inherits the above two methods from the `Node` superclass. They set the shape's `layoutX` and `layoutY` properties, respectively.

These properties represent different things for different `Shapes`. For example, for `Rectangles`, `layoutX` and `layoutY` represent the location of the **top left corner** of the `Rectangle`. However, for `Circles` and `Ellipses`, `layoutX` and `layoutY` represent the location of the **center** of the `Shape`. Thus, be careful when using these methods, and make sure you understand that they will act differently based on the `Shape`.

`void setLayoutX(double value)` and `void setLayoutY(double value)`.

The values passed to each of these methods are the coordinates at which we want the `Shape` to be located. So calling the following methods on an already instantiated `Rectangle` called `rectangle` will position `rectangle` at the position (25, 50) on the screen regardless of its previous position. Let's take a look at how that's done.

```
Rectangle rectangle = new Rectangle(100, 100);  //instantiates a new
Rectangle with height and width of 100
```

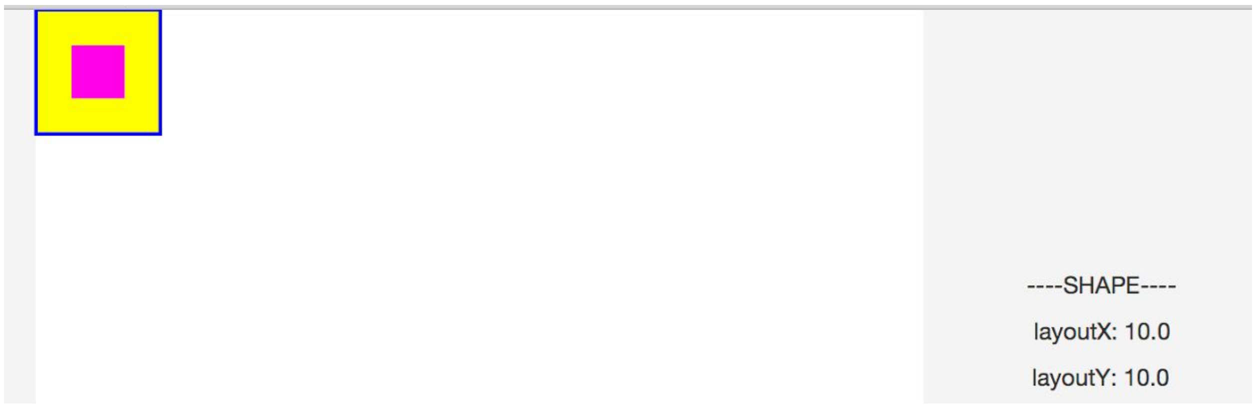
```
rectangle.setLayoutX(25);  //changes the Rectangle's layoutX position
to 25, so the position of rectangle now becomes (25,0)

rectangle.setLayoutY(50);  //changes the Rectangle's layoutY position
to 50, so the new location of rectangle is (25, 50)
```

**Known Issues:**
A very tricky trap in JavaFX is the way `relocate()` and `setLayoutX()` interact when they are used interchangeably.

`setLayoutX/Y` will set the `layoutX` and `layoutY` properties of the object to the desired coordinates. If, however, you choose to give your object a positive stroke (border), meaning that the border will be outside the object, `setLayoutX/Y` will ignore the stroke and re-position the `Shape` based on the top left corner of the actual `Shape`. This is the same issue as with using the `Shape` subclass' methods, so read the **Known Issues** portion of that section to learn about how `Strokes` work.

Unlike `setLayoutX/Y`, `relocate()` includes the stroke when setting the location of the `Shape`, so it will set the top left corner of the entire `Shape` (stroke included) to the given coordinates.

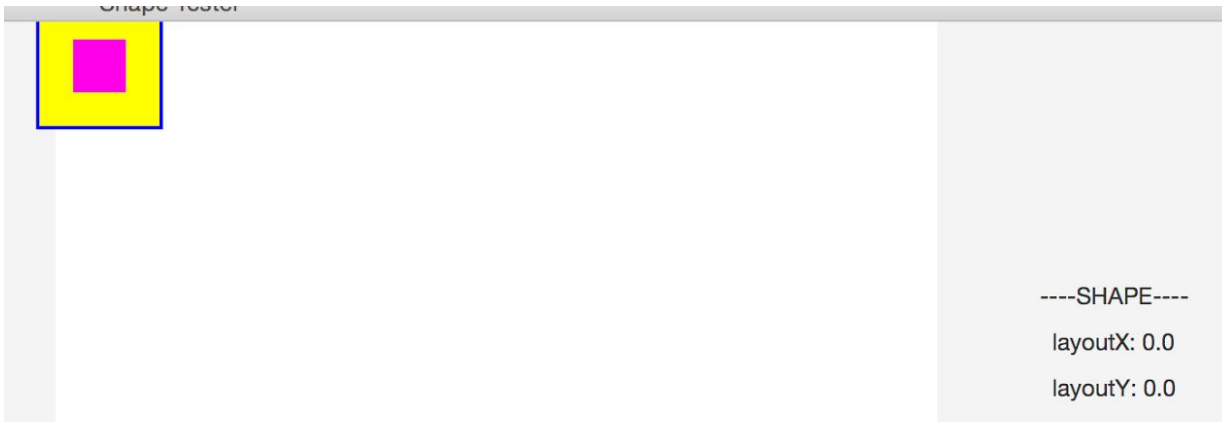
To illustrate the difference between using `relocate()` and `setLayoutX/Y`, suppose we have a `Rectangle` that has a side length of 40 and a stroke (border) of size 20. **Ignore the blue outline**, the stroke we are referring to is the large yellow border of the magenta rectangle. If we call `relocate(0, 0)` on the `Rectangle`, the following would occur:



At this point notice that the top left corner of the entire object (including the stroke) is at location (0,0) while the `layoutX` and `layoutY` properties are both set to 10. The fact that (`layoutX`, `layoutY`) is (10, 10) means that the top left corner of the actual Shape (stroke <u>not</u> included) is located at (10, 10).

**This shows that calling `relocate(x, y)` will not necessarily set the `layoutX` and `layoutY` properties to `x` and `y`. Using `setLayoutX(x)` and `setLayoutY(y)`, however, directly sets the `layoutX` and `layoutY` properties to `x` and `y`.**

Thus, calling `setLayoutX(0)` and `setLayoutY(0)` on the `Rectangle` instead of `relocate(0, 0)` will result in the following scenario:



```
----SHAPE----
layoutX: 0.0
layoutY: 0.0
```

Here, the `layoutX` and `layoutY` properties are actually set to 0, however, part of the Shape's stroke is off the screen (since `setLayoutX/Y` ignores the stroke).

If you decide you want to use `setLayoutX/Y` for setting the locations of your `Shapes`, make sure you fully understand these issues before attempting to use these methods.

Additionally, if used exclusively, the `setLayoutX/Y` methods behave identically to `setTranslateX/Y`, so be sure to read the Known Issues portion of that section.

## Using `setTranslateX()` and `setTranslateY()`

**Description:**

Each `Shape` subclass inherits the following two methods from the `Node` superclass. This includes `Rectangle`, `Ellipse`, `Circle`, `Line`, etc..

`setTranslateX(double value)` and `setTranslateY(double value)`

The value passed in from its parameter defines the x coordinate of the translation that is added to the `Shape`. For example:

```
Rectangle rect = new Rectangle(50, 50);  //This creates a rectangle of
width and height of 100 pixels
rect.setLayoutX(100);
rect.setLayoutY(100);  //Now rectangle is located at (100, 100) (See
Figure 1)
```

```
rect.setTranslateX(50);  //This moves rec 50 pixels to the right of
original position, now located at (150, 100)(See Figure 2)
rect.setTranslateX(-50);  //Moves rec 50 pixels to the left of
original position, now located at (50, 100)
```

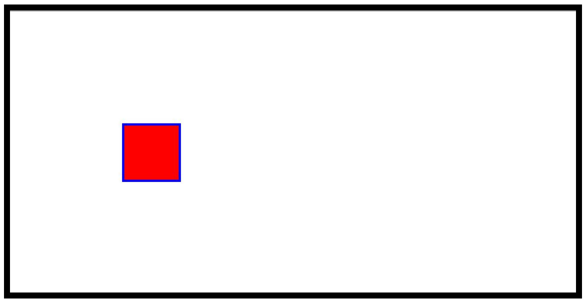


Figure 1. Set at (100, 100)

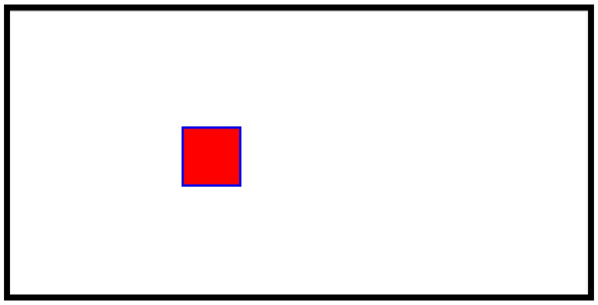


Figure 2. Set at (150, 100)

Note:  this example is purely for pedagogical purposes. **Do not use both `setLayoutX/Y` and `setTranslateX/Y` in your project** -- you won't ever need to and it may produce unwanted results that can be very difficult to debug. Instead, you should choose *either* `setLayoutX/Y` or `setTranslateX/Y` and use that method call for every location-setting need that you have for shapes in your project.

In general, the shape's final translation will be computed as `layoutX` + `translateX` , where `layoutX` establishes the node's stable position and `translateX` optionally makes dynamic adjustments to that position.

For subclasses that have location setting methods, (such as `Rectangle` and `Ellipse`), the final position would actually be `layoutX + translateX + x` (where `x` is the subclass' location property).

**Known Issues:**
One of the main issues with using `setTranslateX/Y` occurs when using these methods in tandem with other location-setting methods (e.g. `setX(), setLayoutX()` ).  As shown in the description above, calling different location-setting methods on one `Shape` will result in the final on screen location of the `Shape` differing from the location parameters passed into `setTranslateX/Y`.

Additionally, if used exclusively, the `setTranslateX/Y` methods behave identically to `setLayoutX/Y`, so be sure to read the Known Issues portion of that section.

## Setting Colors

All subclasses of the `Shape` class, such as `Rectangle`, `Circle`, `Ellipse` and `Line`, have these three methods: `setFill(Paint value)`, `setStroke(Paint value)` and `setStrokeWidth(double value)`. These three methods are *super* important for the upcoming CS15 projects. Let's take a look at how they work.

`setFill` - fill the interior of the `Shape` with the `Paint` value that is passed in from the parameters. We usually pass in a `Color`, which is a subclass of the `Paint` class.

`setStroke` - defines the color of the stroke that is drawn around the outline of the `Shape`. Again, we usually pass in a `Color`.

`setStrokeWidth` - This defines the width of the border of the `Shape`. A value less than 0.0 will be treated as 0.0.

Example:

```
//Instantiating variables
Rectangle rec = new Rectangle();  //The rectangle is filled with the
color black, has a black outline and stroke width of 1.0 by default

rec.setFill(Color.RED);  //sets rec’s fill to red
rec.setStrokeWidth(1.5);  //sets rec’s stroke width to 1.5
rec.setStroke(Color.BLUE);  //sets rec’s stroke to blue
```

## Timelines

We use `javafx.animation.Timeline` to control events (animations, perhaps?) that happen at regular intervals. For this example, we will assume that we have two items that we want to move at regular intervals - a rocket ship and an alien. The catch is that we want our items to move at staggered intervals, not at the same time - we want the rocket to move, then the alien, and so on. Let's assume we have separate `EventHandler<ActionEvent>` classes, `RocketHandler` and `AlienHandler`, for our two items. First, we instantiate our `Timeline`:

```
Timeline timeline = new Timeline();
```

We attach `EventHandler<ActionEvent>` classes to `Timelines` through the `javafx.animation.KeyFrame` class. In the `KeyFrame` constructor, we pass in a `javafx.util.Duration` object, which defines the length (in time) of the `KeyFrame`, and an `EventHandler<ActionEvent>` object, whose `handle(ActionEvent e)` method will be called at the end of the `KeyFrame`'s duration.

The `Duration` class has static methods such as `seconds(double seconds)` and `millis(double milliseconds)` to create `Duration` objects of different lengths. Here, we create a one-second `KeyFrame` with a `RocketHandler` attached to it:

```
KeyFrame frame1 = new KeyFrame(Duration.seconds(1),
                           new RocketHandler());
```

A `Timeline` may have multiple `KeyFrames` of different lengths and with different handlers attached. We attach an `AlienHandler` to a new 50-millisecond `KeyFrame`:

```
KeyFrame frame2 = new KeyFrame(Duration.millis(50),
                          new AlienHandler());
```

Now, we add our `KeyFrames` to the `Timeline`:

```
timeline.getKeyFrames().addAll(frame1, frame2);
// Note that if we only have one KeyFrame we can use add() instead of
// addAll()
```

The `Timeline` will cycle through its `KeyFrames` one after the other. After a one-second delay, `RocketHandler`'s `handle(ActionEvent e)` method will be called, and then, after a 50-millisecond delay, `AlienHandler`'s `handle(ActionEvent e)` method will be called. The `KeyFrame` we add first in the `addAll()` method will come first in the cycle. You may find that in all but the most complicated animations, one `KeyFrame` will be sufficient - multiple `KeyFrames` are useful for events that happen sequentially, but at staggered intervals. For a `Timeline` example with only one keyframe, see the *Graphics* lecture slides.

We must tell our `Timeline` how many times to loop through its sequential list of `KeyFrames` using the `setCycleCount(int cycles)` method. We may specify any number of cycles. If we want the `Timeline` to loop continuously, we may use the built-in constant `Timeline.INDEFINITE`:

```
timeline.setCycleCount(Timeline.INDEFINITE);
```

Finally, we must tell our `Timeline` to start:

```
timeline.play();
```

There is also a `playFromStart()` method to reset the `Timeline` to the start of its `KeyFrame` cycle, and a `stop()` method to stop the `Timeline` and reset it to the start of its `KeyFrame` cycle.

# Keyboard Input

## Handlers and KeyEvents

To make programs react to keyboard input, we need an event handler that responds to `javafx.scene.input.KeyEvent` . We create a class that implements `EventHandler<KeyEvent>` - we'll call ours `KeyHandler`. We must write our `handle(KeyEvent e)` method to respond differently to different keys on the keyboard. We do this using `javafx.scene.input.KeyCode`. A list of `KeyCode` names for every key on the keyboard can be found here.

```
// Note that the KeyEvent will be passed in automatically when the
// method is called
public void handle(KeyEvent keyEvent) {

      switch (keyEvent.getCode()) {

      case <key>:
            <Do something>
            break;

      case <other key>:
            <Do something else>
            break;

      // Etc. etc.

      default:
            break;
      }
}
```

Note that if we use a series of `if-else` statements to check for the different KeyCodes instead of a switch statement, we must write `KeyCode.<key>` instead of just `<key>` when we want to refer to a specific `KeyCode`.

Now that we have written our `handle(KeyEvent keyEvent)` method, we must attach our handler to the `Pane` that contains the elements we wish to control. We assume we have a `Pane` variable `pane` and write:

```
pane.addEventHandler(KeyEvent.KEY_PRESSED, new KeyHandler());
```

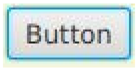
Now, when a key is pressed, the handler's `handle(KeyEvent keyEvent)` method will be called, a `KeyEvent` containing the `KeyCode` of the key pressed will be passed in, and our code will execute!

Note: There is also a `KeyEvent.KEY_RELEASED` constant, which, if passed into the `addEventHandler()` method instead of `KeyEvent.KEY_PRESSED`, will cause the `handle(KeyEvent keyEvent)` method to be called when a key is *released*.

## Focus Control

JavaFX `Nodes` have a property called "focus." If a `Node` (i.e. a `Pane` or a `Button`) is focused, the `Node` *and its child `Nodes`* can receive keyboard input from the user (mouse input still works on unfocused `Nodes`). *Only one `Node` in an application can be focused at a time.* This means that, if a `Button` or other GUI element accidentally becomes focused, keyboard input on a different `Node` (i.e. your `Pane` that requires keyboard input) will not work! Oh no!

In fact, this problem is quite likely to happen. We commonly use the arrow keys for program input, but in JavaFX, the arrow keys also shift the focus between `Nodes`. We could very well find ourselves focusing `Buttons` when we don't want to! We know that a `Button` is focused when it has a blue glow around it:



Luckily, there is a solution. The arrow keys shift the focus between `Nodes` because, when a `KeyEvent` is produced, it "bubbles up" through the program's `Node` hierarchy and causes effects other than what we specify in our `handle(KeyEvent keyEvent)` method. We can prevent the "bubbling up" from happening by calling `keyEvent.consume()` at the very end of our `handle(KeyEvent keyEvent)` method. This "consumes" the `KeyEvent` and prevents it from moving up the `Node` hierarchy and shifting the focus.

The call to `consume()` will solve our problems once our intended `Node` actually *has* the focus, but a `Button` or other `Node` could still unintentionally *start out* with the focus when our program is first run. As it turns out, the *first* `Node` added to the root `Pane` receives the focus when the program begins. Therefore, we should add our `Pane` for which we require keyboard input to our root `Pane` *before* we add any other `Nodes`, especially `Buttons` or `Panes` containing `Buttons`!

Finally, and quite importantly, `Panes` are not focusable by default. Therefore, every time we instantiate a `Pane` for which we want to use keyboard input, we must manually set the `Pane` to be focusable by using the `setFocusTraversable(boolean b)` method. Additionally, although a single `Pane` might be the only focusable `Node` in our program, it is good practice to have that `Pane` request the program's focus explicitly when we instantiate it. We assume that we have a `Pane` called `pane` and write:

```
pane.setFocusTraversable(true);
pane.requestFocus();
```

After we take all these actions, our intended `Pane` will start out with the program’s focus, and our `KeyEvents` will be consumed before they have a chance to shift the focus away from the `Pane`. As a result, the program’s focus will be effectively locked to the intended `Pane`!