

JavaFX Documentation Project

Published 2016-11-06

Table of Contents

1. Introduction	1
1.1. Contributors	1
1.2. Contributing	1
2. Scenegraph	2
3. UI Controls	3
3.1. ListView	3
4. Layout	10
4.1. VBox and HBox	10
4.2. Absolute Positioning with Pane	17
4.3. Clipping	23
4.4. GridPane	28
4.5. GridPane Spanning	32
4.6. GridPane ColumnConstraints and RowConstraints	37
5. CSS	47
6. Performance	48
7. Application Structure	49
8. Best Practices	50
8.1. 1. Styleable Properties	50
8.2. 2. Tasks	50
9. Contributing	58
10. License	59

Chapter 1. Introduction

The JavaFX Documentation Project aims to pull together useful information for JavaFX developers from all over the web. The project is [open source](#) and encourages community participation to ensure that the documentation is as highly polished and useful as possible.

1.1. Contributors

This project would not be possible without the contributors who work hard on the content contained within this documentation. Whenever possible contributors are given attribution within the document when they write a section, but it is also important to gather all names here, at the top of the document, to give the recognition that these contributors deserve.

1.2. Contributing

Contributing to this project is easy - fork the [GitHub](#) repo, edit the relevant files, and create a pull request! Once merged, your content will form a part of the documentation and you'll have the unending appreciation of the entire community!

The JavaFX Documentation Project uses AsciiDoc as the syntax of choice for writing the documentation. The [AsciiDoc Syntax Quick Reference](#) guide is a great resource for those learning how to write AsciiDoc.

Authors are welcome to include a byline beneath the sections that they have authored. To ensure consistency, the recommended format for the byline is the following:

Contributed by <name> - <website>

Chapter 2. Scenegraph

Placeholder whilst things get built...

Chapter 3. UI Controls

3.1. ListView

3.1.1. ListView Filtering in JavaFX

Author: Carl Walker

This article demonstrates how to filter a ListView in a JavaFX Application. Two lists are managed by the Application. One list contains all of the items in the data model. The second list contains the items currently being viewed. A scrap of comparison logic stored as a filter mediates between the two.

Binding is used heavily to keep the data structures in sync with what the user has selected.

This screenshot shows the Application which contains a top row of ToggleButtons which set the filter and a ListView containing the objects.

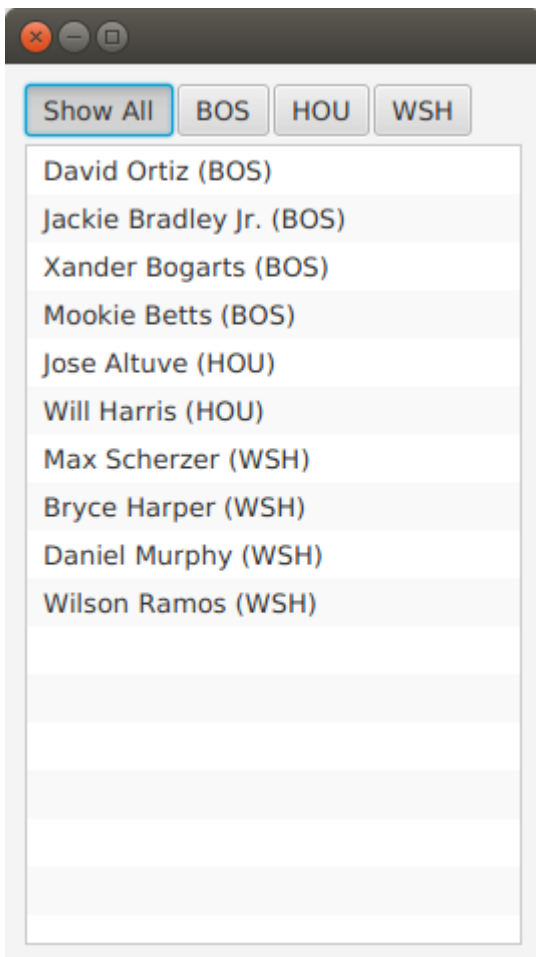


Figure 1. Screenshot of ListView Filtering App

The complete code — a single .java file — is listed at the end of the article.

Data Structures

The program begins with a domain model Player and an array of Player objects.

```

static class Player {

    private final String team;
    private final String playerName;
    public Player(String team, String playerName) {
        this.team = team;
        this.playerName = playerName;
    }
    public String getTeam() {
        return team;
    }
    public String getPlayerName() {
        return playerName;
    }
    @Override
    public String toString() { return playerName + " (" + team + ")"; }
}

```

The Player class contains a pair of fields, team and playerName. A toString() is provided so that when the object is added to the ListView (presented later), a custom ListCell class is not needed.

The test data for this example is a list of American baseball players.

```

Player[] players = {new Player("BOS", "David Ortiz"),
                    new Player("BOS", "Jackie Bradley Jr."),
                    new Player("BOS", "Xander Bogarts"),
                    new Player("BOS", "Mookie Betts"),
                    new Player("HOU", "Jose Altuve"),
                    new Player("HOU", "Will Harris"),
                    new Player("WSH", "Max Scherzer"),
                    new Player("WSH", "Bryce Harper"),
                    new Player("WSH", "Daniel Murphy"),
                    new Player("WSH", "Wilson Ramos") };

```

Model

As mentioned at the start of the article, the ListView filtering is centered around the management of two lists. All the objects are stored in a wrapped ObservableList playersProperty and the objects that are currently viewable are stored in a wrapped FilteredList, viewablePlayersProperty. viewablePlayersProperty is built off of playersProperty so updates made to players that meet the FilteredList criteria will also be made to viewablePlayers.

```
ReadOnlyObjectProperty<ObservableList<Player>> playersProperty =
    new SimpleObjectProperty<>(FXCollections.observableArrayList());

ReadOnlyObjectProperty<FilteredList<Player>> viewablePlayersProperty =
    new SimpleObjectProperty<FilteredList<Player>>(
        new FilteredList<>(playersProperty.get()
        ));
```

`filterProperty()` is a convenience to allow callers to bind to the underlying Predicate.

```
ObjectProperty<Predicate<? super Player>> filterProperty =
    viewablePlayersProperty.get().predicateProperty();
```

The UI root is a VBox which contains an HBox of ToggleButtons and a ListView.

```
VBox vbox = new VBox();
vbox.setPadding( new Insets(10));
vbox.setSpacing(4);

HBox hbox = new HBox();
hbox.setSpacing( 2 );

ToggleGroup filterTG = new ToggleGroup();
```

Filtering Action

A handler is attached the ToggleButtons which will modify `filterProperty`. Each `ToggleButton` is supplied a Predicate in the `userData` field. `toggleHandler` uses this supplied Predicate when setting the filter property. This code sets the special case "Show All" `ToggleButton`.

```
@SuppressWarnings("unchecked")
EventHandler<ActionEvent> toggleHandler = (event) -> {
    ToggleButton tb = (ToggleButton)event.getSource();
    Predicate<Player> filter = (Predicate<Player>)tb.getUserData();
    filterProperty.set( filter );
};

ToggleButton tbShowAll = new ToggleButton("Show All");
tbShowAll.setSelected(true);
tbShowAll.setToggleGroup( filterTG );
tbShowAll.setOnAction(toggleHandler);
tbShowAll.setUserData( (Predicate<Player>) (Player p) -> true);
```

The `ToggleButtons` that filter a specific team are created at runtime based on the `Players` array. This Stream does the following.

1. Distill the list of Players down to a distinct list of team Strings
2. Create a ToggleButton for each team String
3. Set a Predicate for each ToggleButton to be used as a filter
4. Collect the ToggleButtons for addition into the HBox container

```
List<ToggleButton> tbs = Arrays.asList( players)
    .stream()
    .map( (p) -> p.getTeam() )
    .distinct()
    .map( (team) -> {
        ToggleButton tb = new ToggleButton( team );
        tb.setToggleGroup( filterTG );
        tb.setOnAction( toggleHandler );
        tb.setUserData( (Predicate<Player>) (Player p) -> team.equals(p.getTeam())
    );
        return tb;
    })
    .collect(Collectors.toList());

hbox.getChildren().add( tbShowAll );
hbox.getChildren().addAll( tbs );
```

ListView

The next step creates the ListView and binds the ListView to the viewablePlayersProperty. This enables the ListView to receive updates based on the changing filter.

```
ListView<Player> lv = new ListView<>();
lv.itemsProperty().bind( viewablePlayersProperty );
```

The remainder of the program creates a Scene and shows the Stage. onShown loads the data set into the playersProperty and the viewablePlayersProperty lists. Although both lists are in sync in this particular version of the program, if the stock filter is every different than "no filter", this code would not need to be modified.

```
vbox.getChildren().addAll( hbox, lv );

Scene scene = new Scene(vbox);

primaryStage.setScene( scene );
primaryStage.setOnShown((evt) -> {
    playersProperty.get().addAll( players );
});

primaryStage.show();
```


This article used binding to tie a list of viewable Player objects to a ListView. The viewable Players were updated when a ToggleButton is selected. The selection applied a filter to a full set of Players which was maintained separately as a FilteredList (thanks @kleopatra_jx). Binding was used to keep the UI in sync and to allow for a separation of concerns in the design.

Further Reading

To see how such a design would implement basic add and remove functionality, visit the following page https://courses.bekwam.net/public_tutorials/bkcourse_filterlistapp.php.

Complete Code

The code can be tested in a single .java file.

```
public class FilterListApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        //
        // Test data
        //
        Player[] players = {new Player("BOS", "David Ortiz"),
                            new Player("BOS", "Jackie Bradley Jr."),
                            new Player("BOS", "Xander Bogarts"),
                            new Player("BOS", "Mookie Betts"),
                            new Player("HOU", "Jose Altuve"),
                            new Player("HOU", "Will Harris"),
                            new Player("WSH", "Max Scherzer"),
                            new Player("WSH", "Bryce Harper"),
                            new Player("WSH", "Daniel Murphy"),
                            new Player("WSH", "Wilson Ramos") };

        //
        // Set up the model which is two lists of Players and a filter criteria
        //
        ReadOnlyObjectProperty<ObservableList<Player>> playersProperty =
            new SimpleObjectProperty<>(FXCollections.observableArrayList());

        ReadOnlyObjectProperty<FilteredList<Player>> viewablePlayersProperty =
            new SimpleObjectProperty<FilteredList<Player>>(
                new FilteredList<>(playersProperty.get())
            ));

        ObjectProperty<Predicate<? super Player>> filterProperty =
            viewablePlayersProperty.get().predicateProperty();

        //
        // Build the UI
    }
}
```

```

//
VBox vbox = new VBox();
vbox.setPadding( new Insets(10));
vbox.setSpacing(4);

HBox hbox = new HBox();
hbox.setSpacing( 2 );

ToggleGroup filterTG = new ToggleGroup();

//
// The toggleHandler action wills set the filter based on the TB selected
//
@SuppressWarnings("unchecked")
EventHandler<ActionEvent> toggleHandler = (event) -> {
    ToggleButton tb = (ToggleButton)event.getSource();
    Predicate<Player> filter = (Predicate<Player>)tb.getUserData();
    filterProperty.set( filter );
};

ToggleButton tbShowAll = new ToggleButton("Show All");
tbShowAll.setSelected(true);
tbShowAll.setToggleGroup( filterTG );
tbShowAll.setOnAction(toggleHandler);
tbShowAll.setUserData( (Predicate<Player>) (Player p) -> true);

//
// Create a distinct list of teams from the Player objects, then create
// ToggleButtons
//
List<ToggleButton> tbs = Arrays.asList( players)
    .stream()
    .map( (p) -> p.getTeam() )
    .distinct()
    .map( (team) -> {
        ToggleButton tb = new ToggleButton( team );
        tb.setToggleGroup( filterTG );
        tb.setOnAction( toggleHandler );
        tb.setUserData( (Predicate<Player>) (Player p) ->
team.equals(p.getTeam()) );
        return tb;
    })
    .collect(Collectors.toList());

hbox.getChildren().add( tbShowAll );
hbox.getChildren().addAll( tbs );

//
// Create a ListView bound to the viewablePlayers property
//
ListView<Player> lv = new ListView<>();

```

```

lv.itemsProperty().bind( viewablePlayersProperty );

vbox.getChildren().addAll( hbox, lv );

Scene scene = new Scene(vbox);

primaryStage.setScene( scene );
primaryStage.setOnShown((evt) -> {
    playersProperty.get().addAll( players );
});

primaryStage.show();
}

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

static class Player {

    private final String team;
    private final String playerName;
    public Player(String team, String playerName) {
        this.team = team;
        this.playerName = playerName;
    }
    public String getTeam() {
        return team;
    }
    public String getPlayerName() {
        return playerName;
    }
    @Override
    public String toString() { return playerName + " (" + team + ")"; }
}
}

```

Chapter 4. Layout

4.1. VBox and HBox

Author: Carl Walker

Layout in JavaFX begins with selecting the right container controls. The two layout controls I use most often are `VBox` and `HBox`. `VBox` is a container that arranges its children in a vertical stack. `HBox` arranges its children in a horizontal row. The power of these two controls comes from wrapping them and setting a few key properties: alignment, hgrow, and vgrow.

This article will demonstrate these controls through a sample project. A mockup of the project shows a UI with the following:

- A row of top controls containing a Refresh `Button` and a Sign Out `Hyperlink`,
- A `TableView` that will grow to take up the extra vertical space, and
- A Close `Button`.

The UI also features a `Separator` which divides the top part of the screen with what may become a standard lower panel (Save `Button`, Cancel `Button`, etc) for the application.

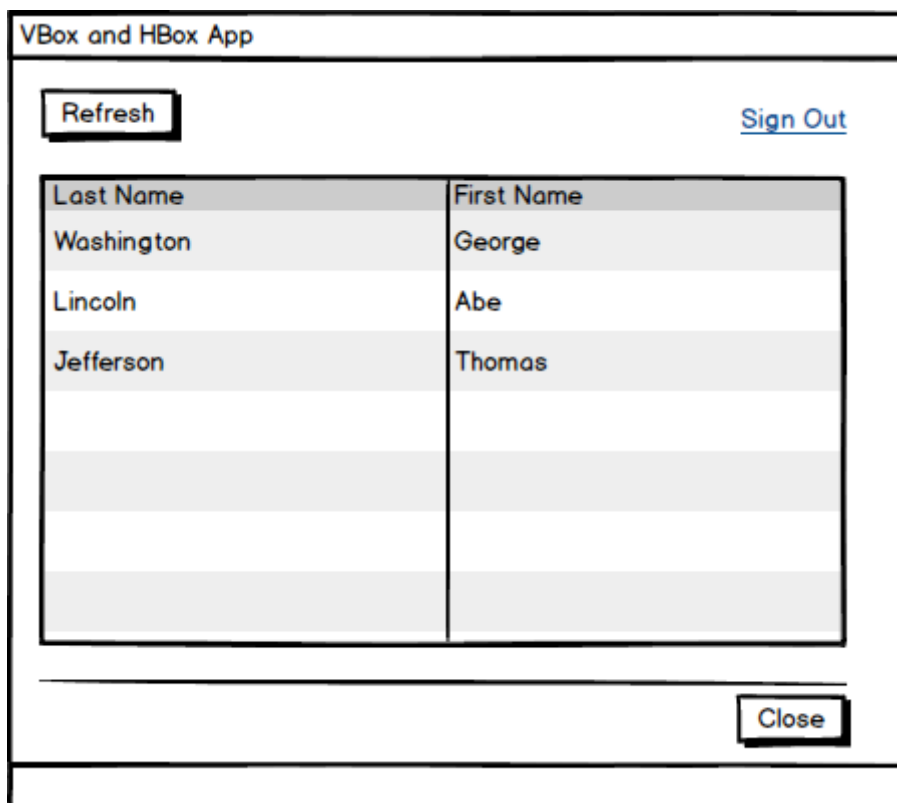


Figure 2. Mockup

4.1.1. Structure

A `VBox` is the outermost container "vbox". This will be the `Parent` provided to the Scene. Simply putting UI controls in this `VBox` will allow the controls—most notably the `TableView`—to stretch to fit the available horizontal space. The top controls, the Refresh `Button` and the Sign Out `Hyperlink`,

are wrapped in an **HBox**. Similarly, I wrap the bottom Close **Button** in an **HBox**, allowing for additional Buttons.

```
VBox vbox = new VBox();

Button btnRefresh = new Button("Refresh");

HBox topRightControls = new HBox();
topRightControls.getChildren().add( signOutLink );

topControls.getChildren().addAll( btnRefresh, topRightControls );

TableView<Customer> tblCustomers = new TableView<>();
Separator sep = new Separator();

HBox bottomControls = new HBox();

Button btnClose = new Button("Close");

bottomControls.getChildren().add( btnClose );

vbox.getChildren().addAll(
    topControls,
    tblCustomers,
    sep,
    bottomControls
);
```

This picture shows the mockup broken down by container. The parent **VBox** is the outermost blue rectangle. The **HBoxes** are the inner rectangles (red and green).

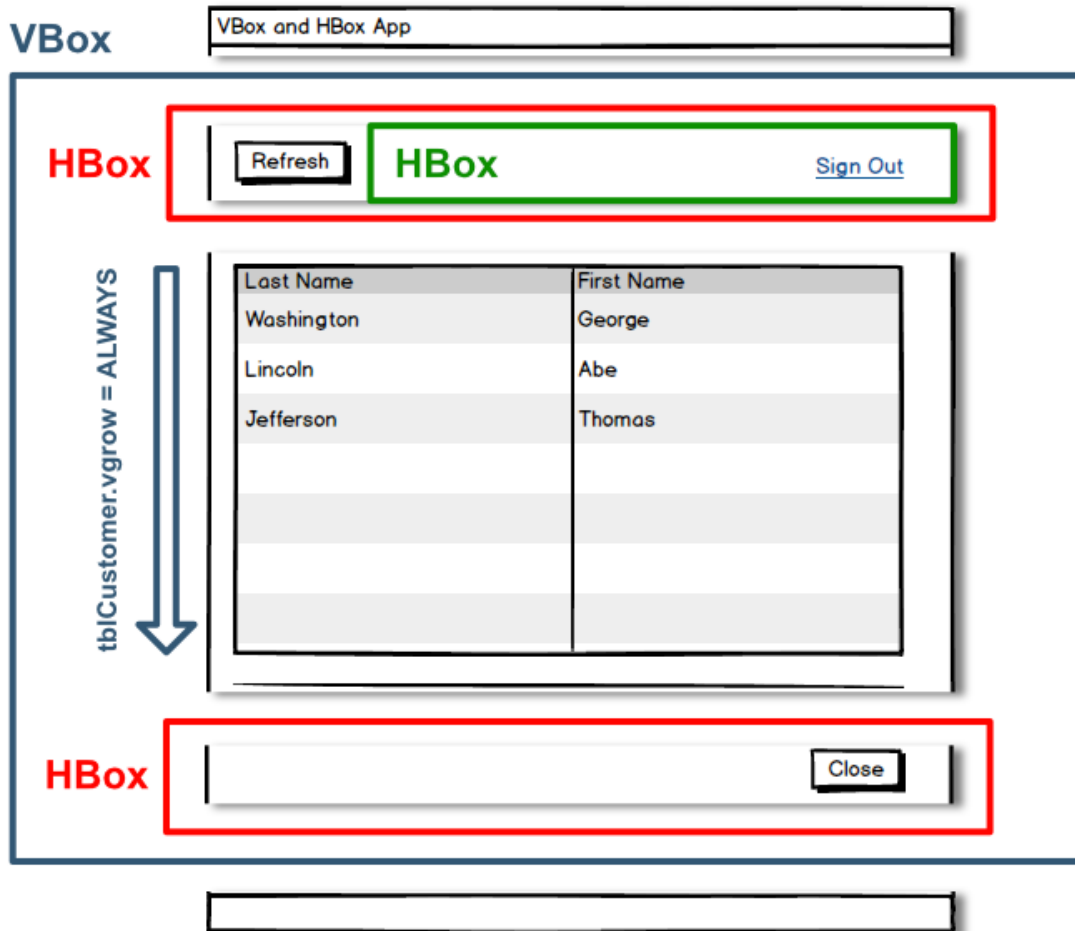


Figure 3. Mockup Broken Down

4.1.2. Alignment and Hgrow

The Refresh **Button** is aligned to the left while the Sign Out **Hyperlink** is aligned to the right. This is accomplished using two **HBoxes**. `topControls` is an **HBox** that contains the Refresh **Button** and also contains an **HBox** with the Sign Out **Hyperlink**. As the screen grows wider, the Sign Out **Hyperlink** will be pulled to the right while the Refresh **Button** will retain its left alignment.

Alignment is the property that tells a container where to position a control. `topControls` sets alignment to the `BOTTOM_LEFT`. `topRightControls` sets alignment to the `BOTTOM_RIGHT`. "BOTTOM" makes sure that the baseline of the text "Refresh" matches the baseline of the text "Sign Out".

In order to make the Sign Out **Hyperlink** move to the right when the screen gets wider, `Priority.ALWAYS` is needed. This is a cue to the JavaFX to widen `topRightControls`. Otherwise, `topControls` will keep the space and `topRightControls` will appear to the left. Sign Out **Hyperlink** still would be right-aligned but in a narrower container.

Notice that `setHgrow()` is a static method and neither invoked on the `topControls` **HBox** nor on itself, `topRightControls`. This is a facet of the JavaFX API that can be confusing because most of the API sets properties via setters on objects.

```
topControls.setAlignment( Pos.BOTTOM_LEFT );

HBox.setHgrow(topRightControls, Priority.ALWAYS );
topRightControls.setAlignment( Pos.BOTTOM_RIGHT );
```

Close **Button** is wrapped in an **HBox** and positioned using the **BOTTOM_RIGHT** priority.

```
bottomControls.setAlignment(Pos.BOTTOM_RIGHT );
```

4.1.3. Vgrow

Since the outermost container is **VBox**, the child **TableView** will expand to take up extra horizontal space when the window is widened. However, vertically resizing the window will produce a gap at the bottom of the screen. The **VBox** does not automatically resize any of its children. As with the **topRightControls HBox**, a grow indicator can be set. In the case of the **HBox**, this was a horizontal resizing instruction **setHgrow()**. For the **TableView** container **VBox**, this will be **setVgrow()**.

```
VBox.setVgrow( tblCustomers, Priority.ALWAYS );
```

4.1.4. Margin

There are a few ways to space out UI controls. This article uses the margin property on several of the containers to add whitespace around the controls. These are set individually rather than using a spacing on the **VBox** so that the **Separator** will span the entire width.

```
VBox.setMargin( topControls, new Insets(10.0d) );
VBox.setMargin( tblCustomers, new Insets(0.0d, 10.0d, 10.0d, 10.0d) );
VBox.setMargin( bottomControls, new Insets(10.0d) );
```

The **Insets** used by **tblCustomers** omits any top spacing to keep the spacing even. JavaFX does not consolidate whitespace as in web design. If the top **Inset** were set to **10.0d** for the **TableView**, the distance between the top controls and the **TableView** would be twice as wide as the distance between any of the other controls.

Notice that these are static methods like the **Priority**.

This picture shows the application when run in its initial 800x600 size.

4.1.5. Select the Right Containers

The philosophy of JavaFX layout is the same as the philosophy of Swing. Select the right container for the task at hand. This article presented the two most versatile containers: **VBox** and **HBox**. By setting properties like alignment, hgrow, and vgrow, you can build incredibly complex layouts through nesting. These are the containers I use the most and often are the only containers that I need.

4.1.6. Complete Code

The code can be tested in a pair of .java files. There is a POJO for the Customer object used by the **TableView**

```
public class Customer {

    private String firstName;
    private String lastName;

    public Customer(String firstName,
                    String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

This is the completed JavaFX **Application** subclass and main.

```
public class VBoxAndHBoxApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        VBox vbox = new VBox();
```

```

HBox topControls = new HBox();
VBox.setMargin( topControls, new Insets(10.0d) );
topControls.setAlignment( Pos.BOTTOM_LEFT );

Button btnRefresh = new Button("Refresh");

HBox topRightControls = new HBox();
HBox.setHgrow(topRightControls, Priority.ALWAYS );
topRightControls.setAlignment( Pos.BOTTOM_RIGHT );
Hyperlink signOutLink = new Hyperlink("Sign Out");
topRightControls.getChildren().add( signOutLink );

topControls.getChildren().addAll( btnRefresh, topRightControls );

TableView<Customer> tblCustomers = new TableView<>();
tblCustomers.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
VBox.setMargin( tblCustomers, new Insets(0.0d, 10.0d, 10.0d, 10.0d) );
VBox.setVgrow( tblCustomers, Priority.ALWAYS );

TableColumn<Customer, String> lastNameCol = new TableColumn<>("Last Name");
lastNameCol.setCellValueFactory(new PropertyValueFactory<>("lastName"));

TableColumn<Customer, String> firstNameCol = new TableColumn<>("First Name");
firstNameCol.setCellValueFactory(new PropertyValueFactory<>("firstName"));

tblCustomers.getColumns().addAll( lastNameCol, firstNameCol );

Separator sep = new Separator();

HBox bottomControls = new HBox();
bottomControls.setAlignment(Pos.BOTTOM_RIGHT );
VBox.setMargin( bottomControls, new Insets(10.0d) );

Button btnClose = new Button("Close");

bottomControls.getChildren().add( btnClose );

vbox.getChildren().addAll(
    topControls,
    tblCustomers,
    sep,
    bottomControls
);

Scene scene = new Scene(vbox );

primaryStage.setScene( scene );
primaryStage.setWidth( 800 );
primaryStage.setHeight( 600 );
primaryStage.setTitle("VBox and HBox App");
primaryStage.setOnShown( (evt) -> loadTable(tblCustomers) );

```

```

        primaryStage.show();
    }

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

    private void loadTable(Table<Customer> tblCustomers) {
        tblCustomers.getItems().add(new Customer("George", "Washington"));
        tblCustomers.getItems().add(new Customer("Abe", "Lincoln"));
        tblCustomers.getItems().add(new Customer("Thomas", "Jefferson"));
    }
}

```

4.2. Absolute Positioning with Pane

Author: Carl Walker

Containers like `VBox` or `BorderPane` align and distribute their children. The superclass `Pane` is also a container, but does not impose an order on its children. The children position themselves through properties like `x`, `centerX`, and `layoutX`. This is called absolute positioning and it is a technique to place a `Shape` or a `Node` at a certain location on the screen.

This screenshot shows an About View. The About View contains a `Hyperlink` in the middle of the screen "About this App". The About View uses several JavaFX shapes to form a design which is cropped to appear like a business card.

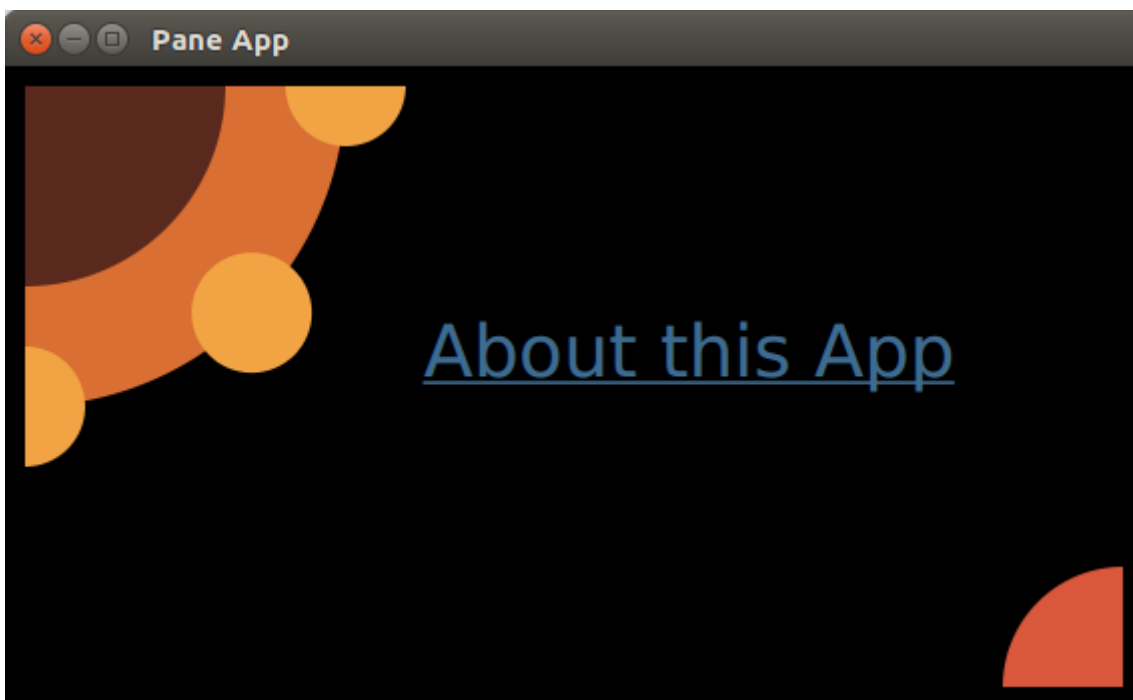


Figure 6. Screenshot of About View in PaneApp

4.2.1. Pane Size

Unlike most containers, **Pane** resizes to fit its contents and not the other way around. This picture is a screenshot from Scenic View taken prior to adding the lower-right **Arc**. The **Pane** is the yellow highlighted area. Notice that it does not take up the full **Stage**.

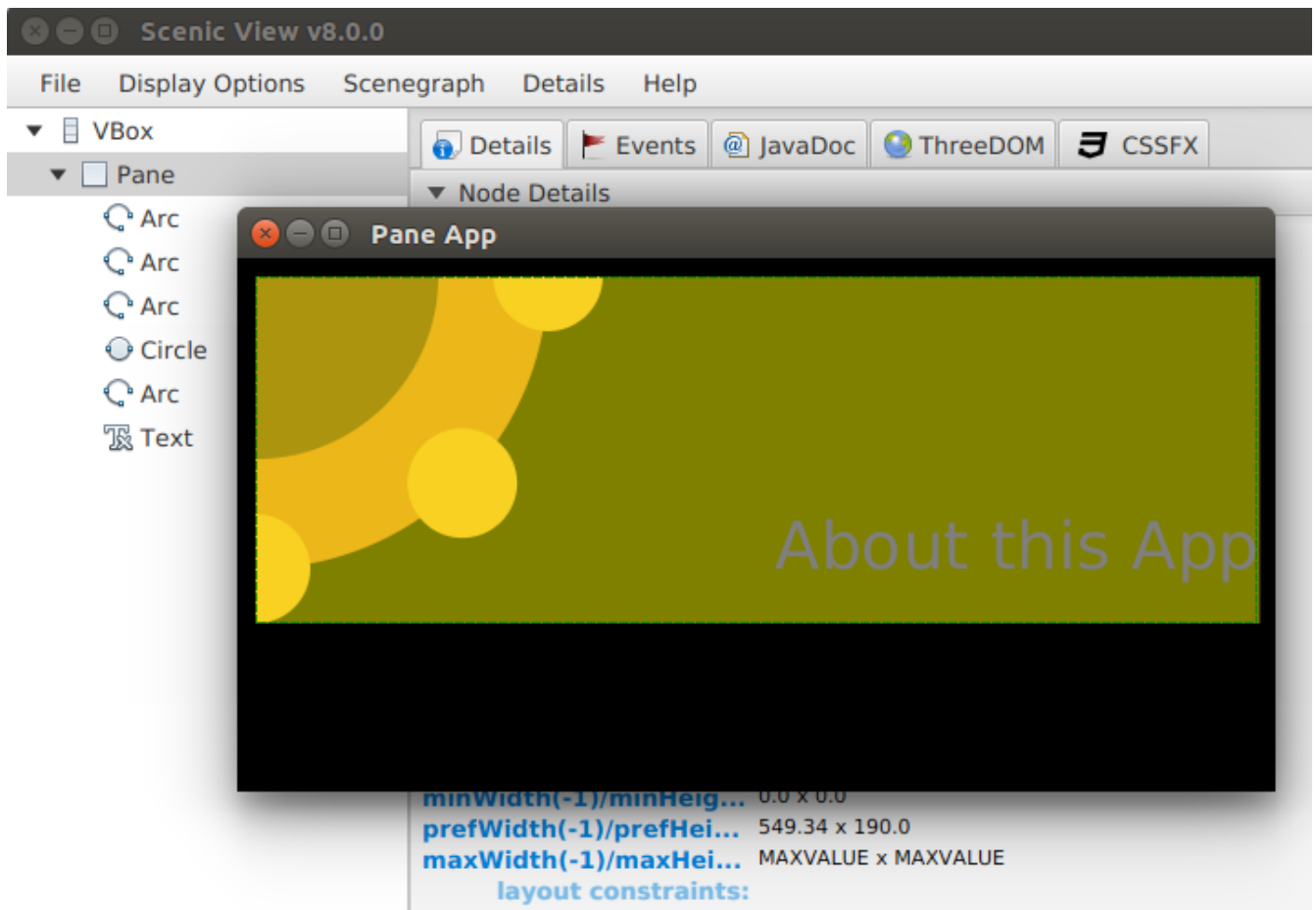


Figure 7. Scenic View Highlighting Partially Built Screen

This is a screenshot taken after the lower-right **Arc** was added. This **Arc** was placed closer to the bottom-right edge of the **Stage**. This forces the **Pane** to stretch to accommodate the expanded contents.

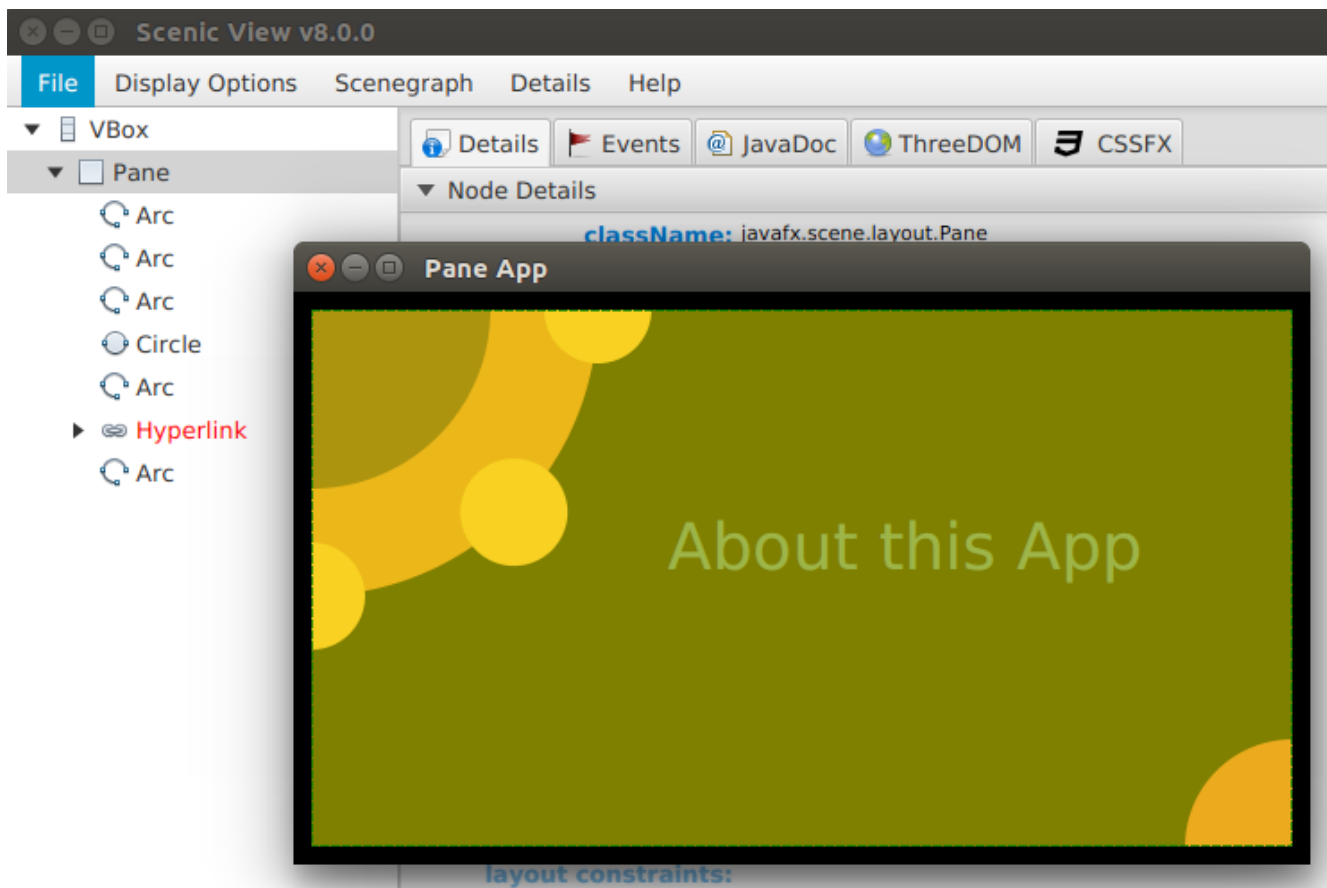


Figure 8. Scenic View Highlighting Expanded Pane

4.2.2. The Pane

The outermost container of the About View is a **VBox** whose sole contents are the **Pane**. The **VBox** is used to fit the entire **Stage** and provides a background.

```
VBox vbox = new VBox();
vbox.setPadding( new Insets( 10 ) );
vbox.setBackground(
    new Background(
        new BackgroundFill(Color.BLACK, new CornerRadii(0), new Insets(0))
    ));

Pane p = new Pane();
```

4.2.3. The Shapes

In the upper left of the screen, there is a group of 4 'Arcs' and 1 'Circle'. This code positions largeArc at (0,0) through the centerX and centerY arguments in the **Arc** constructor. Notice that backgroundArc is also positioned at (0,0) and appears underneath largeArc. **Pane** does not attempt to deconflict overlapping shapes and in this case, overlapping is what is wanted. smArc1 is placed at (0,160) which is down on the Y axis. smArc2 is positioned at (160,0) which is right on the X axis. smCircle is positioned at the same distance as smArc1 and smArc2, but at a 45 degree angle.

```
Arc largeArc = new Arc(0, 0, 100, 100, 270, 90);
largeArc.setType(ArcType.ROUND);

Arc backgroundArc = new Arc(0, 0, 160, 160, 270, 90 );
backgroundArc.setType( ArcType.ROUND );

Arc smArc1 = new Arc( 0, 160, 30, 30, 270, 180);
smArc1.setType(ArcType.ROUND);

Circle smCircle = new Circle(160/Math.sqrt(2.0), 160/Math.sqrt(2.0),
30,Color.web("0xF2A444"));

Arc smArc2 = new Arc( 160, 0, 30, 30, 180, 180);
smArc2.setType(ArcType.ROUND);
```

The lower-right **Arc** is positioned based on the overall height of the **Stage**. The 20 subtracted from the height is the 10 pixel **Insets** from the **VBox** (10 for left + 10 for right).

```
Arc medArc = new Arc(568-20, 320-20, 60, 60, 90, 90);
medArc.setType(ArcType.ROUND);

primaryStage.setWidth( 568 );
primaryStage.setHeight( 320 );
```

4.2.4. The Hyperlink

The **Hyperlink** is positioned offset the center (284,160) which is the width and height of the **Stage** both divided by two. This positions the text of the **Hyperlink** in the lower-right quadrant of the screen, so an offset is needed based on the **Hyperlink** width and height. The dimensions are not available for the **Hyperlink** until the screen is shown, so I make a post-shown adjustment to the position.

```
Hyperlink hyperlink = new Hyperlink("About this App");

primaryStage.setOnShown( (evt) -> {
    hyperlink.setLayoutX( 284 - (hyperlink.getWidth()/3) );
    hyperlink.setLayoutY( 160 - hyperlink.getHeight() );
});
```

The **Hyperlink** is not placed in the true center of the screen. The layoutX value is based on a divide-by-three operation that moves it away from the upper-left design.

4.2.5. Z-Order

As mentioned earlier, **Pane** supports overlapping children. This picture shows the About View with depth added to the upper-left design. The smaller **Arcs** and **Circle** hover over backgroundArc as

does largeArc.

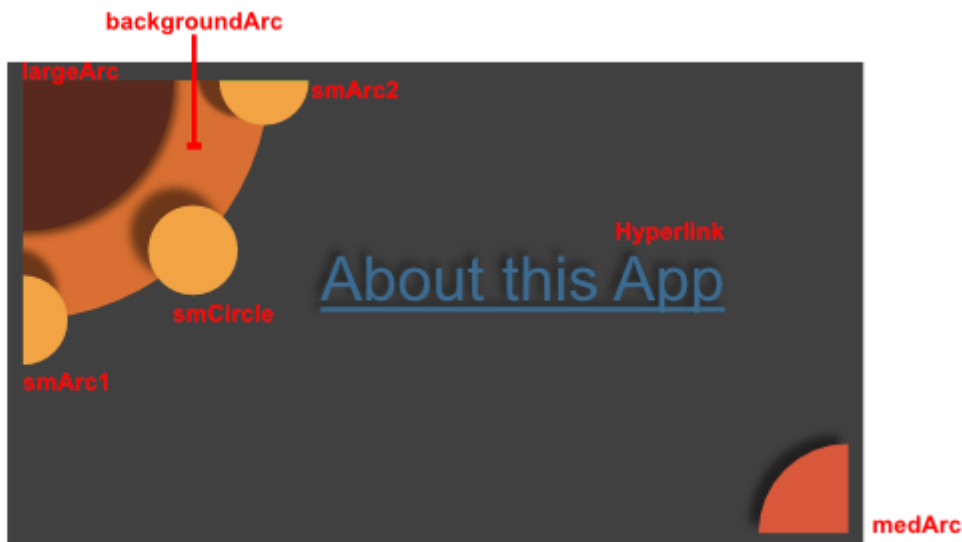


Figure 9. About View Showing Depth

The z-order in this example is determined by the order in which the children are added to the `Pane`. `backgroundArc` is obscured by items added later, most notably `largeArc`. To rearrange the children, use the `toFront()` and `toBack()` methods after the items have been added to the `Pane`.

```
p.getChildren().addAll( backgroundArc, largeArc, smArc1, smCircle, smArc2, hyperlink,
medArc );

vbox.getChildren().add( p );
```

When starting JavaFX, it is tempting to build an absolute layout. Be aware that absolute layouts are brittle, often breaking when the screen is resized or when items are added during the software maintenance phase. Yet, there are good reasons for using absolute positioning. Gaming is one such usage. In a game, you can adjust the (x,y) coordinate of a 'Shape' to move a game piece around the screen. This article demonstrated the JavaFX class `Pane` which provides absolute positioning to any shape-driven UI.

4.2.6. Completed Code

This is the completed JavaFX `Application` subclass and main.

```
public class PaneApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        VBox vbox = new VBox();
        vbox.setPadding( new Insets( 10 ) );
        vbox.setBackground(
```

```

        new Background(
            new BackgroundFill(Color.BLACK, new CornerRadii(0), new Insets(0))
        ));

Pane p = new Pane();

Arc largeArc = new Arc(0, 0, 100, 100, 270, 90);
largeArc.setFill(Color.web("0x59291E"));
largeArc.setType(ArcType.ROUND);

Arc backgroundArc = new Arc(0, 0, 160, 160, 270, 90 );
backgroundArc.setFill( Color.web("0xD96F32") );
backgroundArc.setType( ArcType.ROUND );

Arc smArc1 = new Arc( 0, 160, 30, 30, 270, 180);
smArc1.setFill(Color.web("0xF2A444"));
smArc1.setType(ArcType.ROUND);

Circle smCircle = new Circle(
    160/Math.sqrt(2.0), 160/Math.sqrt(2.0), 30,Color.web("0xF2A444")
);

Arc smArc2 = new Arc( 160, 0, 30, 30, 180, 180);
smArc2.setFill(Color.web("0xF2A444"));
smArc2.setType(ArcType.ROUND);

Hyperlink hyperlink = new Hyperlink("About this App");
hyperlink.setFont( Font.font(36) );
hyperlink.setTextFill( Color.web("0x3E6C93") );
hyperlink.setBorder( Border.EMPTY );

Arc medArc = new Arc(568-20, 320-20, 60, 60, 90, 90);
medArc.setFill(Color.web("0xD9583B"));
medArc.setType(ArcType.ROUND);

p.getChildren().addAll( backgroundArc, largeArc, smArc1, smCircle,
    smArc2, hyperlink, medArc );

vbox.getChildren().add( p );

Scene scene = new Scene(vbox);
scene.setFill(Color.BLACK);

primaryStage.setTitle("Pane App");
primaryStage.setScene( scene );
primaryStage.setWidth( 568 );
primaryStage.setHeight( 320 );
primaryStage.setOnShown( (evt) -> {
    hyperlink.setLayoutX( 284 - (hyperlink.getWidth()/3) );
    hyperlink.setLayoutY( 160 - hyperlink.getHeight() );
});

```



```
        primaryStage.show();
    }

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

4.3. Clipping

Author: Christoph Nahr

Most JavaFX layout containers (base class [Region](#)) automatically position and size their children, so clipping any child contents that might protrude beyond the container's layout bounds is never an issue. The big exception is [Pane](#), a direct subclass of [Region](#) and the base class for all layout containers with publicly accessible children. Unlike its subclasses [Pane](#) does not attempt to arrange its children but simply accepts explicit user positioning and sizing.

This makes [Pane](#) suitable as a drawing surface, similar to [Canvas](#) but rendering user-defined [Shape](#) children rather than direct drawing commands. The problem is that drawing surfaces are usually expected to automatically clip their contents at their bounds. [Canvas](#) does this by default but [Pane](#) does not. From the last paragraph of the Javadoc entry for [Pane](#):

[Pane](#) does not clip its content by default, so it is possible that children's bounds may extend outside its own bounds, either if children are positioned at negative coordinates or the pane is resized smaller than its preferred size.

This quote is somewhat misleading. Children are rendered (wholly or partially) outside their parent [Pane](#) 'whenever' their combination of position and size extends beyond the parent's bounds, regardless of whether the position is negative or the [Pane](#) is ever resized. Quite simply, [Pane](#) only provides a coordinate shift to its children, based on its upper-left corner – but its layout bounds are completely ignored while rendering children. Note that the Javadoc for all [Pane](#) subclasses (that I checked) includes a similar warning. They don't clip their contents either, but as mentioned above this is not usually a problem for them because they automatically arrange their children.

So to properly use [Pane](#) as a drawing surface for [Shapes](#), we need to manually clip its contents. This is somewhat complex, especially when a visible border is involved. I wrote a small demo application to illustrate the default behavior and various steps to fix it. You can download it as [PaneDemo.zip](#) which contains a project for NetBeans 8.2 and Java SE 8u112. The following sections explain each step with screenshots and pertinent code snippets.

4.3.1. Default Behavior

Starting up, [PaneDemo](#) shows what happens when you put an [Ellipse](#) shape into a [Pane](#) that's too small to contain it entirely. The [Pane](#) has a nice thick rounded [Border](#) to visualize its area. The application window is resizable, with the [Pane](#) size following the window size. The three buttons on

the left are used to switch to the other steps in the demo; click Default (Alt+D) to revert to the default output from a later step.

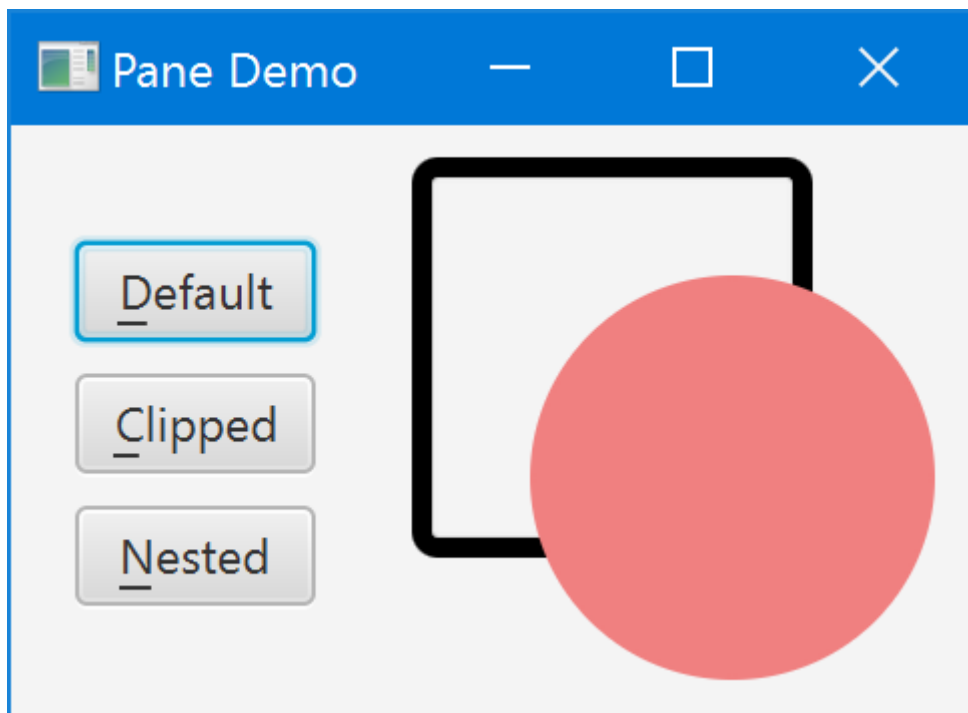


Figure 10. Child Extending Outside Pane Bounds

As you can see, the `Ellipse` overwrites its parent's `Border` and protrudes well beyond it. The following code is used to generate the default view. It's split into several smaller methods, and a constant for the `Border` corner radius, because they will be referenced in the next steps.

```

static final double BORDER_RADIUS = 4;

static Border createBorder() {
    return new Border(
        new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID,
            new CornerRadii(BORDER_RADIUS), BorderStroke.THICK));
}

static Shape createShape() {
    final Ellipse shape = new Ellipse(50, 50);
    shape.setCenterX(80);
    shape.setCenterY(80);
    shape.setFill(Color.LIGHTCORAL);
    shape.setStroke(Color.LIGHTCORAL);
    return shape;
}

static Region createDefault() {
    final Pane pane = new Pane(createShape());
    pane.setBorder(createBorder());
    pane.setPrefSize(100, 100);
    return pane;
}

```

4.3.2. Simple Clipping

Surprisingly, there is no predefined option to have a resizable `Region` automatically clip its children to its current size. Instead, you need to use the basic `clipProperty` defined on `Node` and keep it updated manually to reflect changing layout bounds. Method `clipChildren` below show how this works (with Javadoc because you may want to reuse it in your own code):

```

/**
 * Clips the children of the specified {@link Region} to its current size.
 * This requires attaching a change listener to the region's layout bounds,
 * as JavaFX does not currently provide any built-in way to clip children.
 *
 * @param region the {@link Region} whose children to clip
 * @param arc the {@link Rectangle#arcWidth} and {@link Rectangle#arcHeight}
 *            of the clipping {@link Rectangle}
 * @throws NullPointerException if {@code region} is {@code null}
 */
static void clipChildren(Region region, double arc) {

    final Rectangle outputClip = new Rectangle();
    outputClip.setArcWidth(arc);
    outputClip.setArcHeight(arc);
    region.setClip(outputClip);

    region.layoutBoundsProperty().addListener((ov, oldValue, newValue) -> {
        outputClip.setWidth(newValue.getWidth());
        outputClip.setHeight(newValue.getHeight());
    });
}

static Region createClipped() {
    final Pane pane = new Pane(createShape());
    pane.setBorder(createBorder());
    pane.setPrefSize(100, 100);

    // clipped children still overwrite Border!
    clipChildren(pane, 3 * BORDER_RADIUS);

    return pane;
}

```

Choose Clipped (Alt+C) in PaneDemo to render the corresponding output. Here's how that looks:

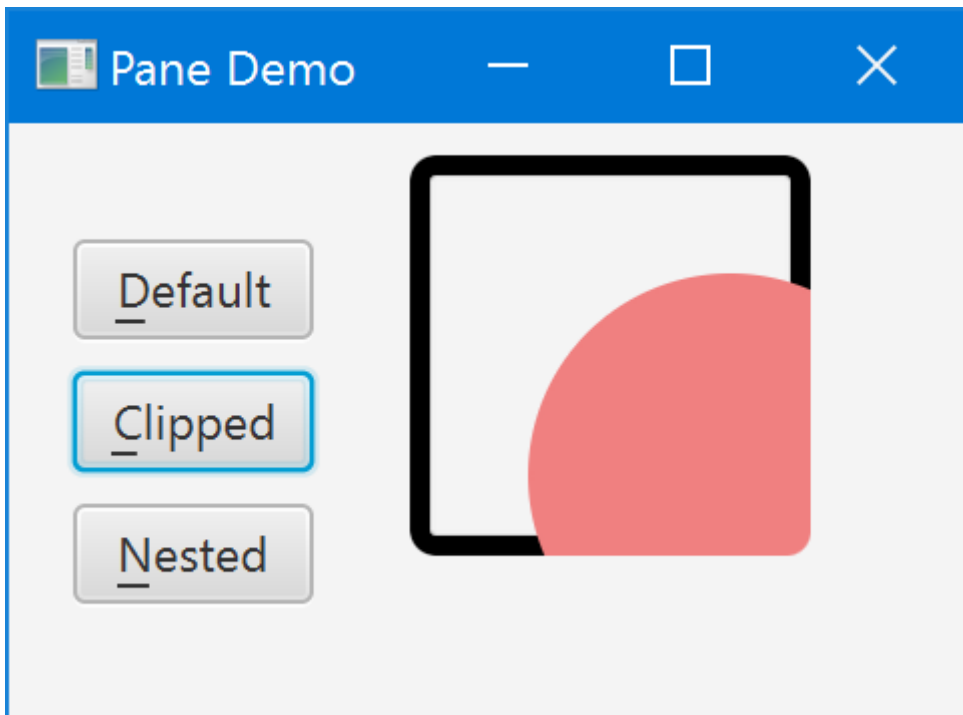


Figure 11. Pane with Clip Applied

That's better. The `Ellipse` no longer protrudes beyond the `Pane` – but still overwrites its `Border`. Also note that we had to manually specify an estimated corner rounding for the clipping `Rectangle` in order to reflect the rounded `Border` corners. This estimate is $3 * \text{BORDER_RADIUS}$ because the corner radius specified on `Border` actually defines its inner radius, and the outer radius (which we need here) will be greater depending on the `Border` thickness. (You could compute the outer radius exactly if you really wanted to, but I skipped that for the demo application.)

4.3.3. Nested Panes

Can we somehow specify a clipping region that excludes a visible `Border`? Not on the drawing `Pane` itself, as far as I can tell. The clipping region affects the `Border` as well as other contents, so if you were to shrink the clipping region to exclude it you would no longer see any `Border` at all. Instead, the solution is to create two nested panes: one inner drawing `Pane` without `Border` that clips exactly to its bounds, and one outer `StackPane` that defines the visible `Border` and also resizes the drawing `Pane`. Here is the final code:

```
static Region createNested() {
    // create drawing Pane without Border or size
    final Pane pane = new Pane(createShape());
    clipChildren(pane, BORDER_RADIUS);

    // create sized enclosing Region with Border
    final Region container = new StackPane(pane);
    container.setBorder(createBorder());
    container.setPrefSize(100, 100);
    return container;
}
```

Choose Nested (Alt+N) in PaneDemo to render the corresponding output. Now everything looks as it

should:

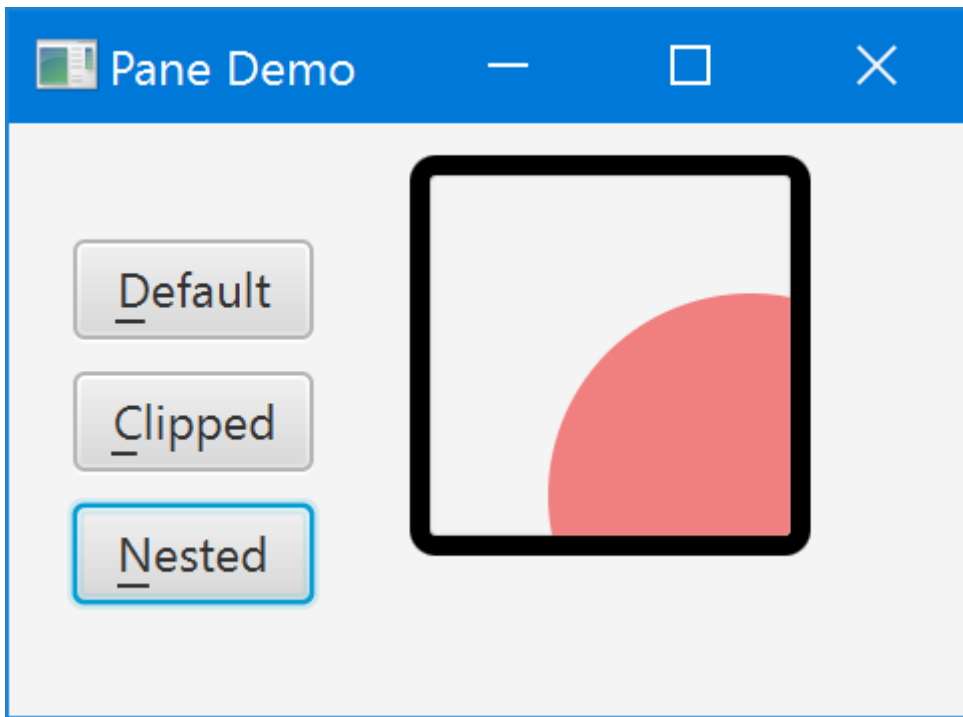


Figure 12. Nesting Pane in StackPane

As an added bonus, we no longer need to guesstimate a correct corner radius for the clipping `Rectangle`. We now clip to the inner rather than outer circumference of our visible `Border`, so we can directly reuse its inner corner radius. Should you specify multiple different corner radii or an otherwise more complex `Border` you'd have to define a correspondingly more complex clipping `Shape`.

There is one small caveat. The top-left corner of the drawing `Pane` to which all child coordinates are relative now starts *within* the visible `Border`. If you retroactively change a single `Pane` with visible `Border` to nested panes as shown here, all children will exhibit a slight positioning shift corresponding to the `Border` thickness.

4.4. GridPane

Author: Carl Walker

Forms in business applications often use a layout that mimics a database record. For each column in a table, a header is added on the left-hand side which is matched with a row value on the right-hand side. JavaFX has a special purpose control called `GridPane` for this type of layout that keeps contents aligned by row and column. `GridPane` also supports spanning for more complex layouts.

This screenshot shows a basic `GridPane` layout. On the left-hand side of the form, there is a column of field names: Email, Priority, Problem, Description. On the right-hand side of the form, there is a column of controls which will display the value of the corresponding field. The field names are of type `Label` and the value controls are a mixture including `TextField`, `TextArea`, and `ComboBox`.

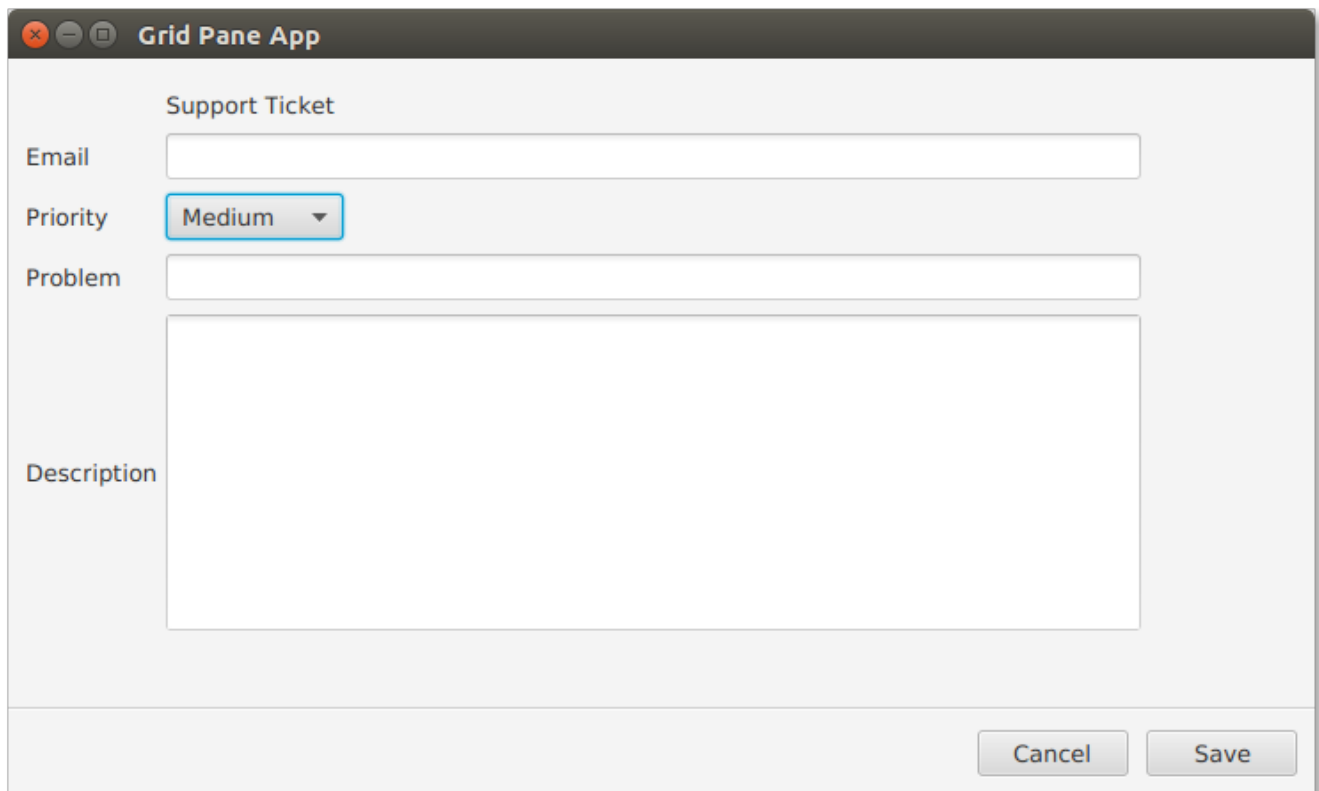


Figure 13. Field Name / Value Pairs in a GridPane

The following code shows the objects created for the form. "vbox" is the root of the **Scene** and will also contain the **ButtonBar** at the base of the form.

```
VBox vbox = new VBox();

GridPane gp = new GridPane();

Label lblTitle = new Label("Support Ticket");

Label lblEmail = new Label("Email");
TextField tfEmail = new TextField();

Label lblPriority = new Label("Priority");
ObservableList<String> priorities = FXCollections.observableArrayList("Medium",
    "High", "Low");
ComboBox<String> cbPriority = new ComboBox<>(priorities);

Label lblProblem = new Label("Problem");
TextField tfProblem = new TextField();

Label lblDescription = new Label("Description");
TextArea taDescription = new TextArea();
```

GridPane has a handy method `setGridLinesVisible()` which shows the grid structure and gutters. It is especially useful in more complex layouts where spanning is involved because gaps in the row/col assignments can cause shifts in the layout.

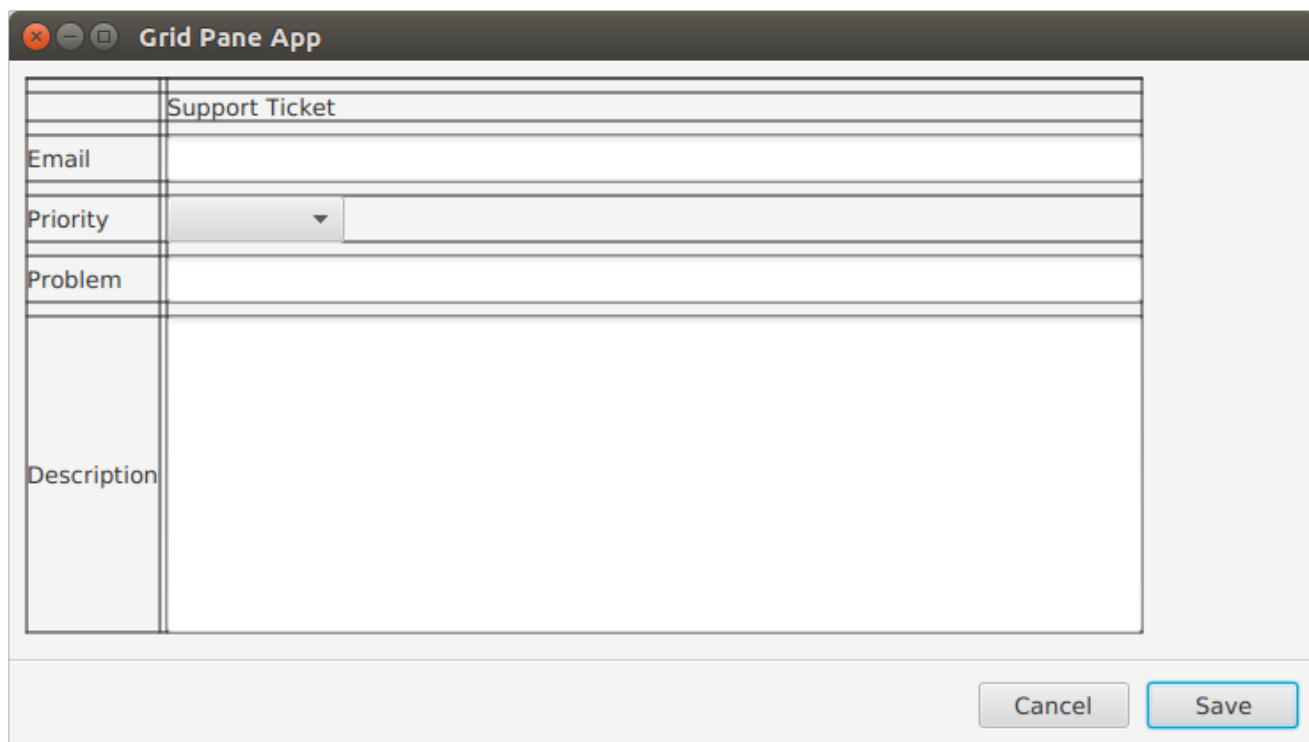


Figure 14. Grid Structure and Gutters

4.4.1. Spacing

As a container, `GridPane` has a padding property that can be set to surround the `GridPane` contents with whitespace. "padding" will take an `Inset` object as a parameter. In this example, 10 pixels of whitespace is applied to all sides so a short form constructor is used for `Inset`.

Within the `GridPane`, `vgap` and `hgap` control the gutters. The `hgap` is set to 4 to keep the fields close to their values. `vgap` is slightly larger to help with mouse navigation.

```
gp.setPadding( new Insets(10) );
gp.setHgap( 4 );
gp.setVgap( 8 );
```

In order to keep the lower part of the form consistent, a `Priority` is set on the `VBox`. This will *not* `resize` the individual rows however. For individual resize specifications, use `ColumnConstraints` and `RowConstraints`.

```
VBox.setVgrow(gp, Priority.ALWAYS );
```

4.4.2. Adding Items

Unlike containers like `BorderPane` or `HBox`, Nodes need to specify their position within the `GridPane`. This is done with the `add()` method on the `GridPane` and not the `add` method on a container children property. This form of the `GridPane` `add()` method takes a zero-based column position and a zero-based row position. This code puts two statements on the same line for readability.


```
gp.add( lblTitle,      1, 1); // empty item at 0,0
gp.add( lblEmail,      0, 2); gp.add(tfEmail,      1, 2);
gp.add( lblPriority,    0, 3); gp.add( cbPriority,    1, 3);
gp.add( lblProblem,    0, 4); gp.add( tfProblem,    1, 4);
gp.add( lblDescription, 0, 5); gp.add( taDescription, 1, 5);
```

lblTitle is put in the second column of the first row. There is no entry in the first column of the first row.

Subsequent additions are presented in pairs. Field name **Label** objects are put in the first column (column index=0) and value controls are put in the second column (column index=1). The rows are added by the incremented second value. For example, lblPriority is put in the fourth row along with its **ComboBox**.

GridPane is an important container in JavaFX business application design. When you have a requirement for name / value pairs, **GridPane** will be an easy way to support the strong column orientation of a traditional form.

4.4.3. Completed Code

The following class is the complete code form the example. This include the definition of the **ButtonBar** which was not presented in the preceding sections focused on **GridPane**.

```
public class GridPaneApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        VBox vbox = new VBox();

        GridPane gp = new GridPane();
        gp.setPadding( new Insets(10) );
        gp.setHgap( 4 );
        gp.setVgap( 8 );

        VBox.setVgrow(gp, Priority.ALWAYS );

        Label lblTitle = new Label("Support Ticket");

        Label lblEmail = new Label("Email");
        TextField tfEmail = new TextField();

        Label lblPriority = new Label("Priority");
        ObservableList<String> priorities =
            FXCollections.observableArrayList("Medium", "High", "Low");
        ComboBox<String> cbPriority = new ComboBox<>(priorities);

        Label lblProblem = new Label("Problem");
        TextField tfProblem = new TextField();
```

```

Label lblDescription = new Label("Description");
TextArea taDescription = new TextArea();

gp.add( lblTitle,      1, 1); // empty item at 0,0
gp.add( lblEmail,      0, 2); gp.add(tfEmail,      1, 2);
gp.add( lblPriority,    0, 3); gp.add( cbPriority,    1, 3);
gp.add( lblProblem,    0, 4); gp.add( tfProblem,    1, 4);
gp.add( lblDescription, 0, 5); gp.add( taDescription, 1, 5);

Separator sep = new Separator(); // hr

ButtonBar buttonBar = new ButtonBar();
buttonBar.setPadding( new Insets(10) );

Button saveButton = new Button("Save");
Button cancelButton = new Button("Cancel");

buttonBar.setButtonData(saveButton, ButtonBar.ButtonData.OK_DONE);
buttonBar.setButtonData(cancelButton, ButtonBar.ButtonData.CANCEL_CLOSE);

buttonBar.getButtons().addAll(saveButton, cancelButton);

vbox.getChildren().addAll( gp, sep, buttonBar );

Scene scene = new Scene(vbox);

primaryStage.setTitle("Grid Pane App");
primaryStage.setScene(scene);
primaryStage.setWidth( 736 );
primaryStage.setHeight( 414 );
primaryStage.show();
}

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

```

4.5. GridPane Spanning

Author: Carl Walker

For more complex forms implemented with **GridPane**, spanning is supported. Spanning allows a control to claim the space of neighboring columns (**colspan**) and neighboring rows (**rowspan**). This screenshot shows a form that extends the example in the previous section. The two-column layout of the earlier version has been replaced with a multiple-column layout. Fields like Problem and Description retain the original structure. But controls were added to the rows that formerly held only Email and Priority.

Support Ticket

Email Contract

Priority Severity Category

Problem

Description

Cancel Save

Figure 15. Spanning Columns

Turning the grid lines on, notice that the former two column grid is replaced with a six column grid. The third row containing six items — 3 field name / value pairs — dictates the structure. The rest of the form will use spanning in order to fill in the whitespace.

Support Ticket

Email Contract

Priority Severity Category

Problem

Description

Cancel Save

Figure 16. Lines Highlighting Spanning

The `VBox` and `GridPane` container objects used in this update follow. There is a little more `Vgap` to help the user select the `ComboBox` controls.

```
GridPane gp = new GridPane();
gp.setPadding( new Insets(10) );
gp.setHgap( 4 );
gp.setVgap( 10 );

VBox.setVgrow(gp, Priority.ALWAYS );
```

These are control creation statements from the updated example.

```
Label lblTitle = new Label("Support Ticket");

Label lblEmail = new Label("Email");
TextField tfEmail = new TextField();

Label lblContract = new Label("Contract");
TextField tfContract = new TextField();

Label lblPriority = new Label("Priority");
ObservableList<String> priorities =
    FXCollections.observableArrayList("Medium", "High", "Low");
ComboBox<String> cbPriority = new ComboBox<>(priorities);

Label lblSeverity = new Label("Severity");
ObservableList<String> severities =
    FXCollections.observableArrayList("Blocker", "Workaround", "N/A");
ComboBox<String> cbSeverity = new ComboBox<>(severities);

Label lblCategory = new Label("Category");
ObservableList<String> categories =
    FXCollections.observableArrayList("Bug", "Feature");
ComboBox<String> cbCategory = new ComboBox<>(categories);

Label lblProblem = new Label("Problem");
TextField tfProblem = new TextField();

Label lblDescription = new Label("Description");
TextArea taDescription = new TextArea();
```

As in the earlier version, the controls are added to the `GridPane` using the `add()` method. A column and row are specified. In this snippet, the indexing is not straightforward as there are gaps expected to be filled in by spanning content.

```

gp.add( lblTitle,      1, 0); // empty item at 0,0

gp.add( lblEmail,      0, 1);
gp.add( tfEmail,       1, 1);
gp.add( lblContract,   4, 1 );
gp.add( tfContract,    5, 1 );

gp.add( lblPriority,    0, 2);
gp.add( cbPriority,     1, 2);
gp.add( lblSeverity,   2, 2);
gp.add( cbSeverity,    3, 2);
gp.add( lblCategory,   4, 2);
gp.add( cbCategory,    5, 2);

gp.add( lblProblem,     0, 3); gp.add( tfProblem,     1, 3);
gp.add( lblDescription, 0, 4); gp.add( taDescription, 1, 4);

```

Finally, the spanning definitions are set using a static method on `GridPane`. There is a similar method to do row spanning. Title will take up 5 columns as will Problem and Description. Email shares a row with Contract, but will take up more columns. The third row of ComboBoxes is a set of three field/value pairs each taking up one column.

```

GridPane.setColumnSpan( lblTitle, 5 );
GridPane.setColumnSpan( tfEmail, 3 );
GridPane.setColumnSpan( tfProblem, 5 );
GridPane.setColumnSpan( taDescription, 5 );

```

Alternatively, a variation of the `add()` method will `columnSpan` and `rowSpan` arguments to avoid the subsequent static method call.

This expanded `GridPane` example demonstrated column spanning. The same capability is available for row spanning which would allow a control to claim additional vertical space. Spanning keeps controls aligned even in cases where the number of items in a given row (or column) varies. To keep the focus on the spanning topic, this grid allowed the column widths to vary. The article on `ColumnConstraints` and `RowConstraints` will focus on building true modular and column typographical grids by better controlling the columns (and rows).

4.5.1. Completed Code

The following is the completed code for the spanning `GridPane` example.

```

public class ComplexGridPaneApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        VBox vbox = new VBox();

```

```

GridPane gp = new GridPane();
gp.setPadding( new Insets(10) );
gp.setHgap( 4 );
gp.setVgap( 10 );

VBox.setVgrow(gp, Priority.ALWAYS );

Label lblTitle = new Label("Support Ticket");

Label lblEmail = new Label("Email");
TextField tfEmail = new TextField();

Label lblContract = new Label("Contract");
TextField tfContract = new TextField();

Label lblPriority = new Label("Priority");
ObservableList<String> priorities =
    FXCollections.observableArrayList("Medium", "High", "Low");
ComboBox<String> cbPriority = new ComboBox<>(priorities);

Label lblSeverity = new Label("Severity");
ObservableList<String> severities =
    FXCollections.observableArrayList("Blocker", "Workaround", "N/A");
ComboBox<String> cbSeverity = new ComboBox<>(severities);

Label lblCategory = new Label("Category");
ObservableList<String> categories = FXCollections.observableArrayList("Bug",
"Feature");
ComboBox<String> cbCategory = new ComboBox<>(categories);

Label lblProblem = new Label("Problem");
TextField tfProblem = new TextField();

Label lblDescription = new Label("Description");
TextArea taDescription = new TextArea();

gp.add( lblTitle,      1, 0); // empty item at 0,0

gp.add( lblEmail,      0, 1);
gp.add( tfEmail,       1, 1);
gp.add( lblContract,   4, 1 );
gp.add( tfContract,    5, 1 );

gp.add( lblPriority,    0, 2);
gp.add( cbPriority,     1, 2);
gp.add( lblSeverity,   2, 2);
gp.add( cbSeverity,    3, 2);
gp.add( lblCategory,   4, 2);
gp.add( cbCategory,    5, 2);

```

```

gp.add( lblProblem,      0, 3); gp.add( tfProblem,      1, 3);
gp.add( lblDescription, 0, 4); gp.add( taDescription, 1, 4);

GridPane.setColumnSpan( lblTitle, 5 );
GridPane.setColumnSpan( tfEmail, 3 );
GridPane.setColumnSpan( tfProblem, 5 );
GridPane.setColumnSpan( taDescription, 5 );

Separator sep = new Separator(); // hr

ButtonBar buttonBar = new ButtonBar();
buttonBar.setPadding( new Insets(10) );

Button saveButton = new Button("Save");
Button cancelButton = new Button("Cancel");

buttonBar.setButtonData(saveButton, ButtonBar.ButtonData.OK_DONE);
buttonBar.setButtonData(cancelButton, ButtonBar.ButtonData.CANCEL_CLOSE);

buttonBar.getButtons().addAll(saveButton, cancelButton);

vbox.getChildren().addAll( gp, sep, buttonBar );

Scene scene = new Scene(vbox);

primaryStage.setTitle("Grid Pane App");
primaryStage.setScene(scene);
primaryStage.setWidth( 736 );
primaryStage.setHeight( 414 );
primaryStage.show();
}

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

```

4.6. GridPane ColumnConstraints and RowConstraints

Author: Carl Walker

Previous articles on [GridPane](#) showed how to create a two-column layout with field names on the left-hand side and field values on the right-hand side. That example was expanded to add more controls to a given row and to use spanning handle gaps in content. This article introduces a pair of JavaFX classes [ColumnConstraints](#) and [RowConstraints](#). These classes give additional specification to a row or column. In this example, a row containing a [TextArea](#) will be given all extra space when the window is resized. The two columns will be set to equals widths.

This screenshot shows an example modified from previous articles. The demo program for this

article has a rotated feel whereby the field names are paired with the field values vertically (on top of the values) rather than horizontally. Row spanning and column spanning is used to align items that are larger than a single cell.

The screenshot shows a window titled "Grid Pane App" with a support ticket form. The form is organized into sections: "Support Ticket", "Email", "Contract", "Priority" (with radio buttons for Medium, High, and Low), "Severity" (with a dropdown menu showing "Workaround"), "Category" (with a dropdown menu showing "Feature"), "Problem", and "Description". Two red rectangles are overlaid on the form, each containing the text "Extra Space Available". One rectangle is positioned to the right of the "Severity" and "Category" dropdowns, and the other is positioned below the "Description" text area. At the bottom of the window are "Cancel" and "Save" buttons.

Figure 17. Example App Using Row and Column Spanning

The red rectangles and text are not part of the UI. They are identifying sections of the screen that will be addressed later with `ColumnConstraints` and `RowConstraints`.

This code is the creation of the `Scene` root and `GridPane` objects.


```

VBox vbox = new VBox();

GridPane gp = new GridPane();
gp.setPadding( new Insets(10) );
gp.setHgap( 4 );
gp.setVgap( 10 );

VBox.setVgrow(gp, Priority.ALWAYS );

```

This code creates the UI controls objects used in the article. Notice that Priority is now implemented as a **VBox** containing RadioButtons.

```

Label lblTitle = new Label("Support Ticket");

Label lblEmail = new Label("Email");
TextField tfEmail = new TextField();

Label lblContract = new Label("Contract");
TextField tfContract = new TextField();

Label lblPriority = new Label("Priority");
RadioButton rbMedium = new RadioButton("Medium");
RadioButton rbHigh = new RadioButton("High");
RadioButton rbLow = new RadioButton("Low");
VBox priorityVBox = new VBox();
priorityVBox.setSpacing( 2 );
GridPane.setVgrow(priorityVBox, Priority.SOMETIMES);
priorityVBox.getChildren().addAll( lblPriority, rbMedium, rbHigh, rbLow );

Label lblSeverity = new Label("Severity");
ObservableList<String> severities =
    FXCollections.observableArrayList("Blocker", "Workaround", "N/A");
ComboBox<String> cbSeverity = new ComboBox<>(severities);

Label lblCategory = new Label("Category");
ObservableList<String> categories =
    FXCollections.observableArrayList("Bug", "Feature");
ComboBox<String> cbCategory = new ComboBox<>(categories);

Label lblProblem = new Label("Problem");
TextField tfProblem = new TextField();

Label lblDescription = new Label("Description");
TextArea taDescription = new TextArea();

```

The Label and value control pairings of Email, Contract, Problem, and Description are put in a single column. They should take the full width of the **GridPane** so each has its columnSpan set to 2.

```
GridPane.setColumnSpan( tfEmail, 2 );
GridPane.setColumnSpan( tfContract, 2 );
GridPane.setColumnSpan( tfProblem, 2 );
GridPane.setColumnSpan( taDescription, 2 );
```

The new Priority RadioButtons are matched horizontally with four controls for Severity and Category. This `rowSpan` setting instructs JavaFX to put the VBox containing the RadioButton in a merged cell that is four rows in height.

```
GridPane.setRowSpan( priorityVBox, 4 );
```

4.6.1. Row Constraints

At this point, the code reflects the UI screenshot presented in [Example App Using Row and Column Spanning](#). To reallocate the extra space at the base of the form, use a `RowConstraints` object to set a `Priority.ALWAYS` on the row of the `TextArea`. This will result in the `TextArea` growing to fill the available space with something usable.

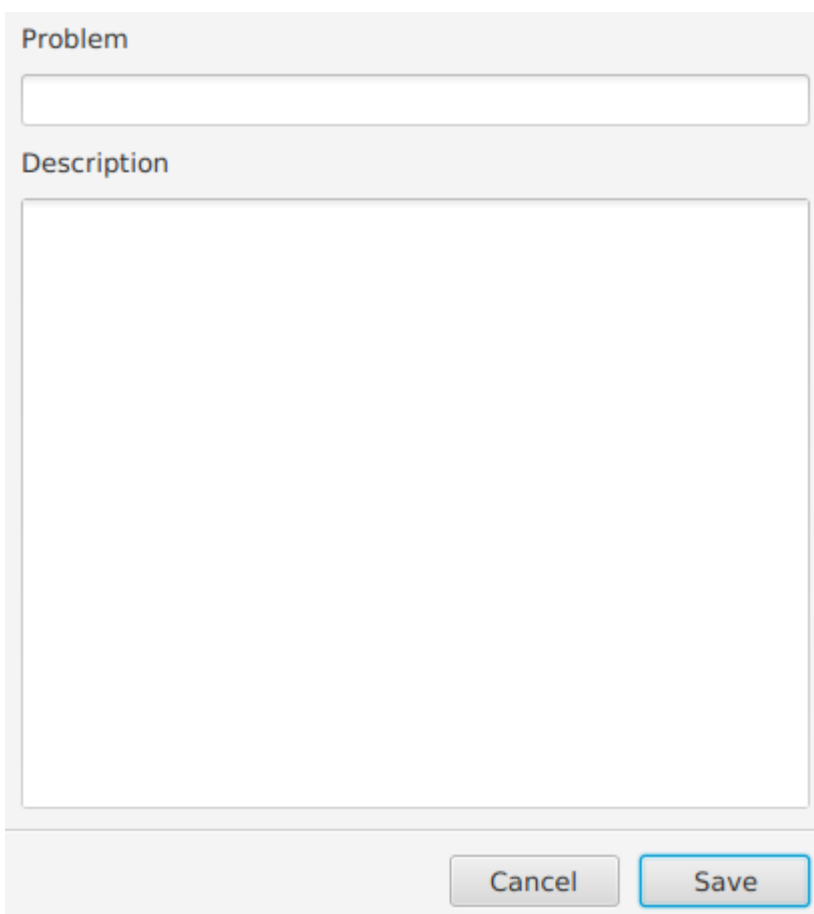


Figure 18. *TextArea Grows to Fill Extra Space*

This code is a `RowConstraints` object to the `GridPane` for the `TextArea`. Prior to the setter, `RowConstraints` objects are allocated for all of the other rows. The `set` method of `getRowConstraints()` will throw an index exception when you specify row 12 without first allocating an object.

```

RowConstraints taDescriptionRowConstraints = new RowConstraints();
taDescriptionRowConstraints.setVgrow(Priority.ALWAYS);

for( int i=0; i<13; i++ ) {
    gp.getRowConstraints().add( new RowConstraints() );
}

gp.getRowConstraints().set( 12, taDescriptionRowConstraints );

```

As an alternative syntax, there is a `setConstraints()` method available from the `GridPane`. This will pass in several values and obviates the need for the dedicated `columnSpan` set call for the `TextArea`. The `RowConstraints` code from the previous listing will not appear in the finished program.

```

gp.setConstraints(taDescription,
                 0, 12,
                 2, 1,
                 HPos.LEFT, VPos.TOP,
                 Priority.SOMETIMES, Priority.ALWAYS);

```

This code identifies the `Node` at (0,12) which is the `TextArea`. The `TextArea` will span 2 columns but only 1 row. The `HPos` and `Vpos` are set to the TOP LEFT. Finally, the `Priority` of the `hgrow` is `SOMETIMES` and the `vgrow` is `ALWAYS`. Since the `TextArea` is the only row with "ALWAYS", it will get the additional space. If there were other `ALWAYS` settings, the space would be shared among multiple rows.

4.6.2. Column Constraints

To properly allocate the space surrounding the Severity and Category controls, `ColumnConstraints` will be specified. The default behavior allocates less space to the first column because of the smaller `Priority` `RadioButtons`. The following wireframe shows the desired layout which has equal columns separated by a gutter of 4 pixels (`Hgap`).

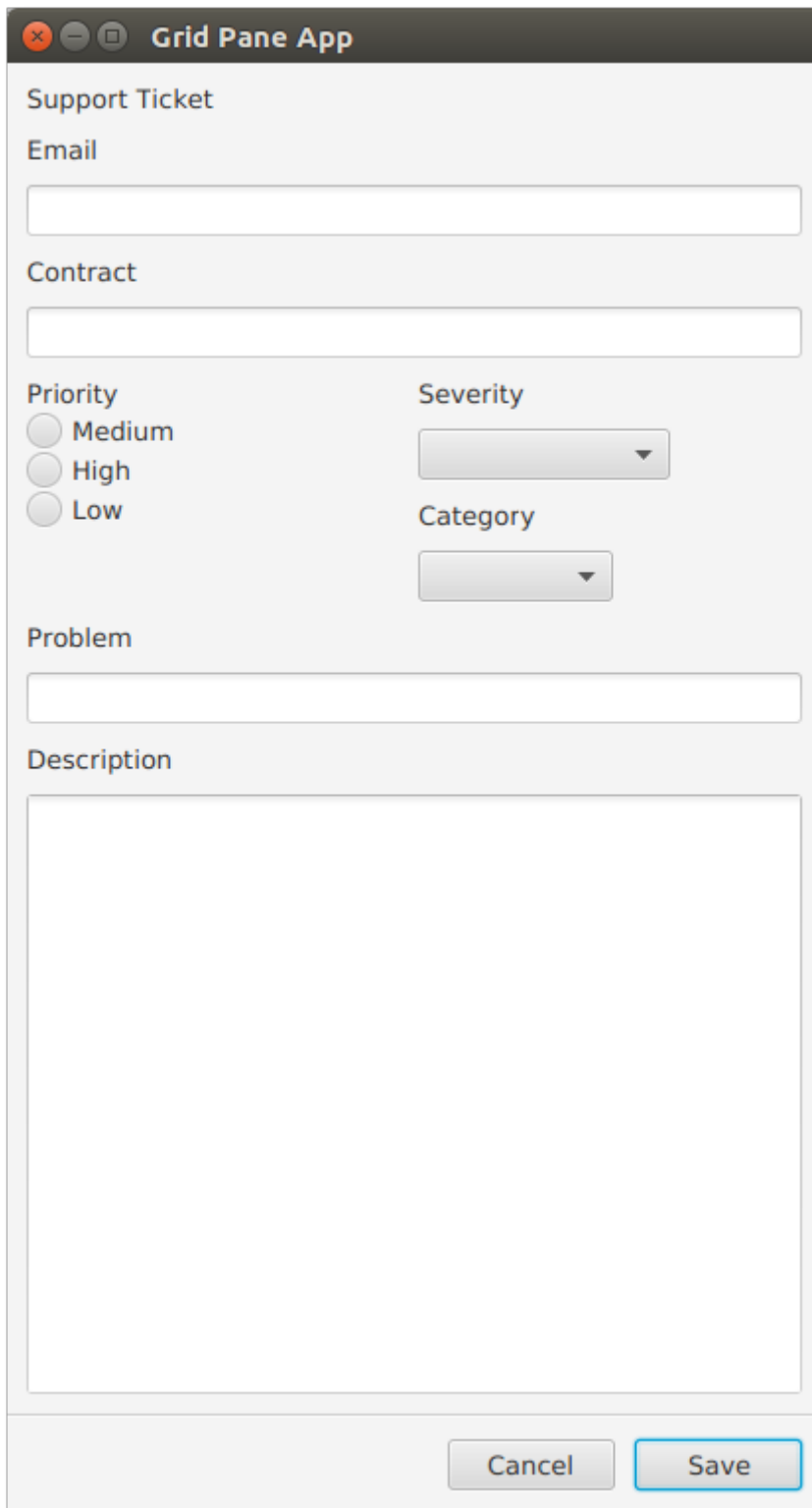
The wireframe shows a window titled "Grid Pane App". Inside, there is a form with two columns. The first column contains labels: "Support Ticket", "Email", "Contract", "Priority", "Problem", and "Description". The second column contains labels: "Severity" and "Category". There are three radio buttons for "Priority" (Medium, High, Low) and two dropdown menus for "Severity" and "Category". At the bottom, there are "Cancel" and "Save" buttons.

Figure 19. Wireframe of the Demo App

To make the column widths equal, define two `ColumnConstraint` objects and use a percentage specifier.

```
ColumnConstraints col1 = new ColumnConstraints();
col1.setPercentWidth( 50 );
ColumnConstraints col2 = new ColumnConstraints();
col2.setPercentWidth( 50 );
gp.getColumnConstraints().addAll( col1, col2 );
```

This is a screenshot of the finished example.



The screenshot shows a JavaFX application window titled "Grid Pane App". The window contains a form for creating a support ticket. The form is organized into sections: "Support Ticket" (header), "Email" (text input), "Contract" (text input), "Priority" (radio buttons for Medium, High, Low), "Severity" (dropdown menu), "Category" (dropdown menu), "Problem" (text input), and "Description" (large text area). At the bottom of the window are "Cancel" and "Save" buttons. The "Save" button is highlighted with a blue border.

Figure 20. App With Extra Space Properly Allocated

GridPane is an important control in developing JavaFX business applications. When working on a requirement involving name / value pairs and a single record view, use **GridPane**. While **GridPane** is easier to use than the **GridBagLayout** from Swing, I still find that the API is a little inconvenient (assigning own indexes, disassociated constraints). Fortunately, there is Scene Builder which simplifies the construction of this form greatly.

4.6.3. Completed Code

The following is the completed code for the constraints GridPane example.

```
public class ConstraintsGridPaneApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        VBox vbox = new VBox();

        GridPane gp = new GridPane();
        gp.setPadding( new Insets(10) );
        gp.setHgap( 4 );
        gp.setVgap( 10 );

        VBox.setVgrow(gp, Priority.ALWAYS );

        Label lblTitle = new Label("Support Ticket");

        Label lblEmail = new Label("Email");
        TextField tfEmail = new TextField();

        Label lblContract = new Label("Contract");
        TextField tfContract = new TextField();

        Label lblPriority = new Label("Priority");
        RadioButton rbMedium = new RadioButton("Medium");
        RadioButton rbHigh = new RadioButton("High");
        RadioButton rbLow = new RadioButton("Low");
        VBox priorityVBox = new VBox();
        priorityVBox.setSpacing( 2 );
        GridPane.setVgrow(priorityVBox, Priority.SOMETIMES);
        priorityVBox.getChildren().addAll( lblPriority, rbMedium, rbHigh, rbLow );

        Label lblSeverity = new Label("Severity");
        ObservableList<String> severities =
FXCollections.observableArrayList("Blocker", "Workaround", "N/A");
        ComboBox<String> cbSeverity = new ComboBox<>(severities);

        Label lblCategory = new Label("Category");
        ObservableList<String> categories = FXCollections.observableArrayList("Bug",
"Feature");
        ComboBox<String> cbCategory = new ComboBox<>(categories);

        Label lblProblem = new Label("Problem");
        TextField tfProblem = new TextField();

        Label lblDescription = new Label("Description");
        TextArea taDescription = new TextArea();
    }
}
```

```

gp.add( lblTitle,      0, 0);

gp.add( lblEmail,      0, 1);
gp.add( tfEmail,       0, 2);

gp.add( lblContract,   0, 3 );
gp.add( tfContract,    0, 4 );

gp.add( priorityVBox,  0, 5);

gp.add( lblSeverity,   1, 5);
gp.add( cbSeverity,    1, 6);
gp.add( lblCategory,   1, 7);
gp.add( cbCategory,    1, 8);

gp.add( lblProblem,    0, 9);
gp.add( tfProblem,     0, 10);

gp.add( lblDescription, 0, 11);
gp.add( taDescription,  0, 12);

GridPane.setColumnSpan( tfEmail, 2 );
GridPane.setColumnSpan( tfContract, 2 );
GridPane.setColumnSpan( tfProblem, 2 );

GridPane.setRowSpan( priorityVBox, 4 );

gp.setConstraints(taDescription,
                  0, 12,
                  2, 1,
                  HPos.LEFT, VPos.TOP,
                  Priority.SOMETIMES, Priority.ALWAYS);

ColumnConstraints col1 = new ColumnConstraints();
col1.setPercentWidth( 50 );
ColumnConstraints col2 = new ColumnConstraints();
col2.setPercentWidth( 50 );
gp.getColumnConstraints().addAll( col1, col2 );

Separator sep = new Separator(); // hr

ButtonBar buttonBar = new ButtonBar();
buttonBar.setPadding( new Insets(10) );

Button saveButton = new Button("Save");
Button cancelButton = new Button("Cancel");

buttonBar.setButtonData(saveButton, ButtonBar.ButtonData.OK_DONE);
buttonBar.setButtonData(cancelButton, ButtonBar.ButtonData.CANCEL_CLOSE);

```

```
        buttonBar.getButtons().addAll(saveButton, cancelButton);

        vbox.getChildren().addAll( gp, sep, buttonBar );

        Scene scene = new Scene(vbox);

        primaryStage.setTitle("Grid Pane App");
        primaryStage.setScene(scene);
        primaryStage.setWidth( 414 );
        primaryStage.setHeight( 736 );
        primaryStage.show();
    }

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


Chapter 5. CSS

Placeholder whilst things get built...

Chapter 6. Performance

Placeholder whilst things get built...

Chapter 7. Application Structure

Placeholder whilst things get built...

Chapter 8. Best Practices

Placeholder whilst things get built...

1. Styleable Properties

8.1. 1. Styleable Properties

Author: Gerrit Grunwald

```
/* Member variables for StyleablePropertyFactory
 * and StyleableProperty
 */
private static final StyleablePropertyFactory<MY_CTRL> FACTORY =
    new StyleablePropertyFactory<>(Control.getClassCssMetaData());

private static final CssMetaData<MY_CTRL, Color> COLOR =
    FACTORY.createColorCssMetaData("-color", s -> s.color, Color.RED, false);
private final StyleableProperty<Color> color = new
SimpleStyleableObjectProperty<>(COLOR, this, "color");

// Getter, Setter and Property method
public Color getColor() {
    return this.color.getValue();
}

public void setColor(final Color color) {
    this.color.setValue(COLOR);
}

public ObjectProperty<Color> colorProperty() {
    return (ObjectProperty<Color>) this.color;
}

// Return CSS Metadata
public static List<CssMetaData<? extends Styleable, ?>> getClassCssMetaData() {
    return FACTORY.getCssMetaData();
}

@Override public List<CssMetaData<? extends Styleable, ?>> getControlCssMetaData() {
    return getClassCssMetaData();
}
```

8.2. 2. Tasks

Author: Carl Walker

This article demonstrates how to use a JavaFX Task to keep the UI responsive. It is imperative that any operation taking more than a few hundred milliseconds be executed on a separate Thread to avoid locking up the UI. A Task wraps up the sequence of steps in a long-running operation and provides callbacks for the possible outcomes.

The **Task** class also keeps the user aware of the operation through properties which can be bound to UI controls like ProgressBars and Labels. The binding dynamically updates the UI. These properties include

1. **runningProperty** - Whether or not the Task is running
2. **progressProperty** - The percent complete of an operation
3. **messageProperty** - Text describing a step in the operation

8.2.1. Demonstration

The following screenshots show the operation of an HTML retrieval application.

Entering a URL and pressing "Go" will start a JavaFX Task. When running, the Task will make an HBox visible that contains a ProgressBar and a Label. The ProgressBar and Label are updated throughout the operation.

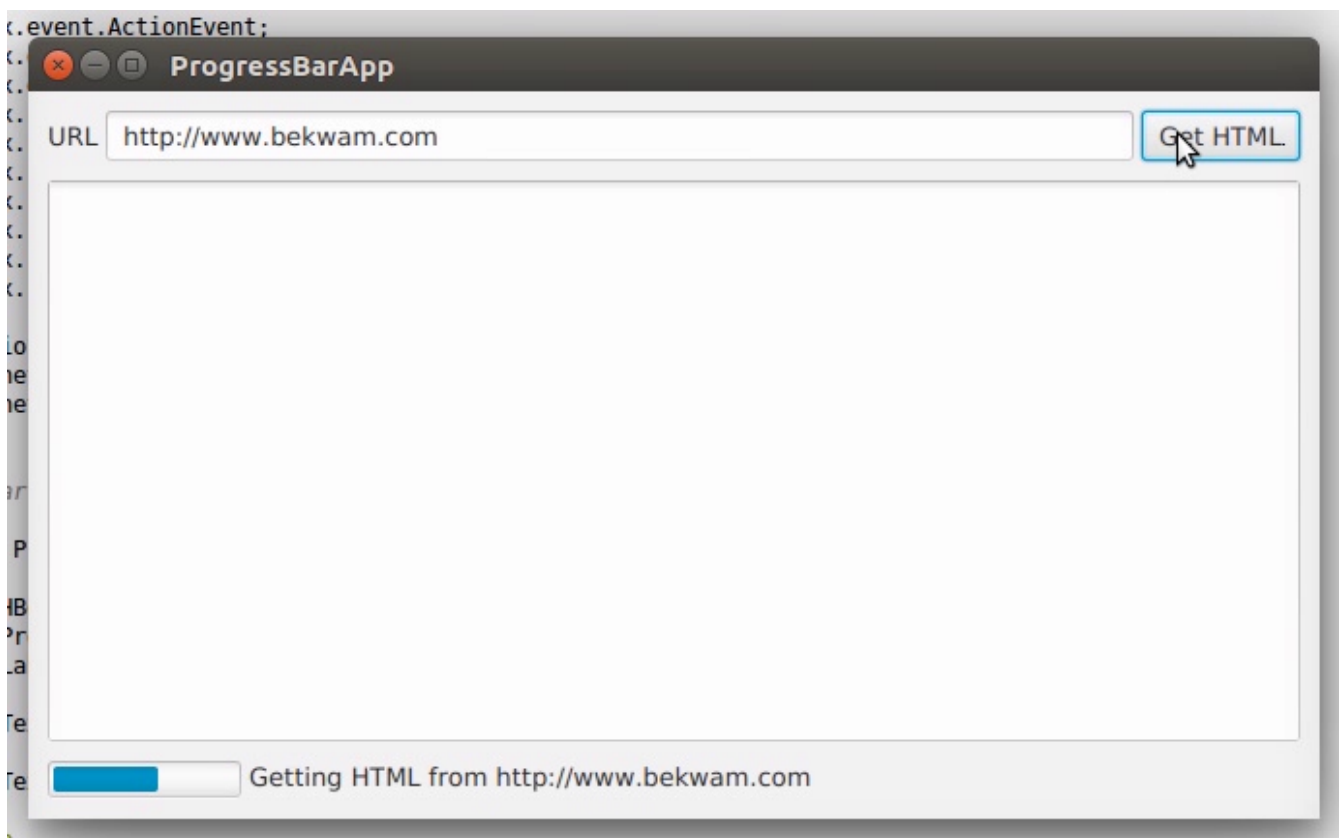


Figure 21. Screenshot of App Showing ProgressBar and Label

When the retrieval is finished, a `succeeded()` callback is invoked and the UI is updated. Note that the `succeeded()` callback takes place on the FX Thread, so it is safe to manipulate controls.

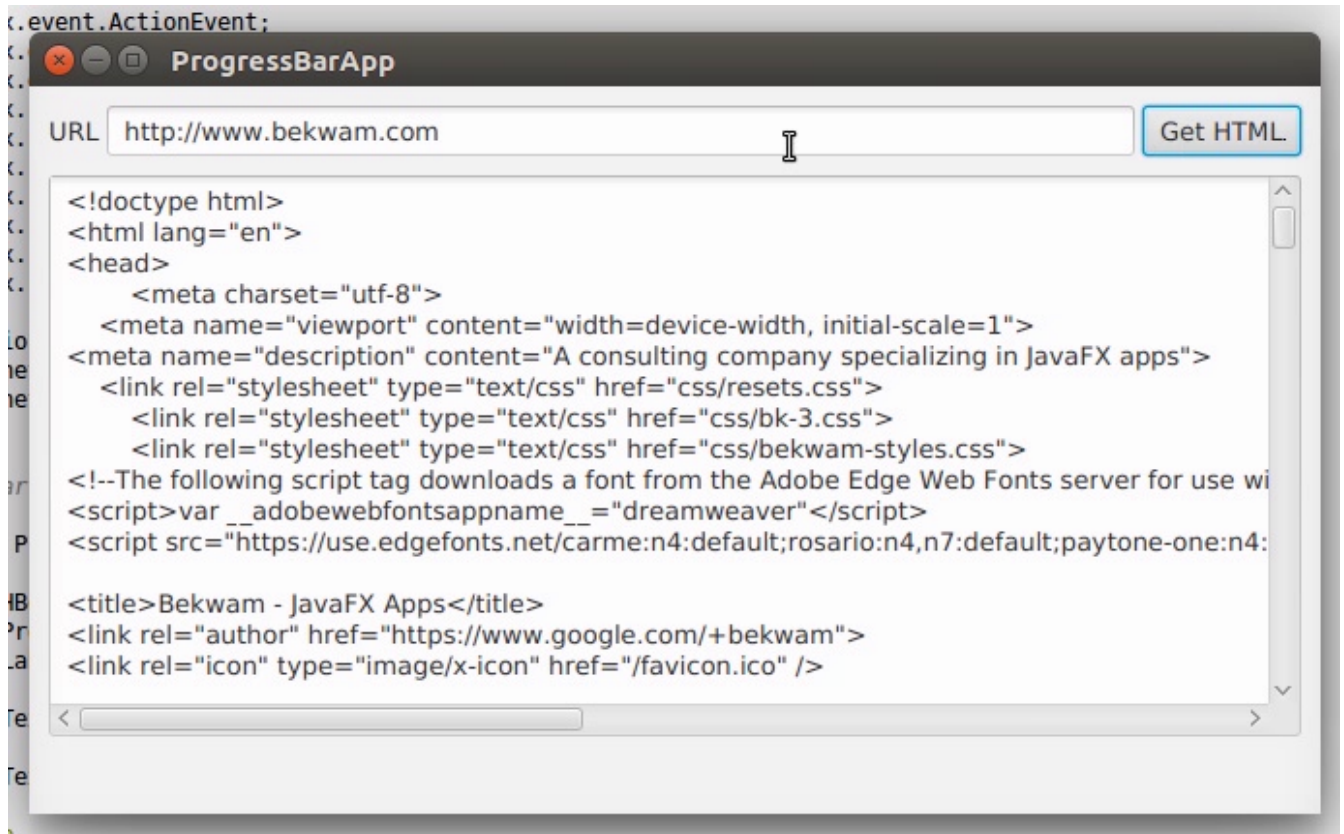


Figure 22. Screenshot of App Showing Successful Retrieval

If there was an error retrieving the HTML, a `failed()` callback is invoked and an error Alert is shown. `failed()` also takes place on the FX Thread. This screenshot shows invalid input. An "h" is used in the URL instead of the correct "http".

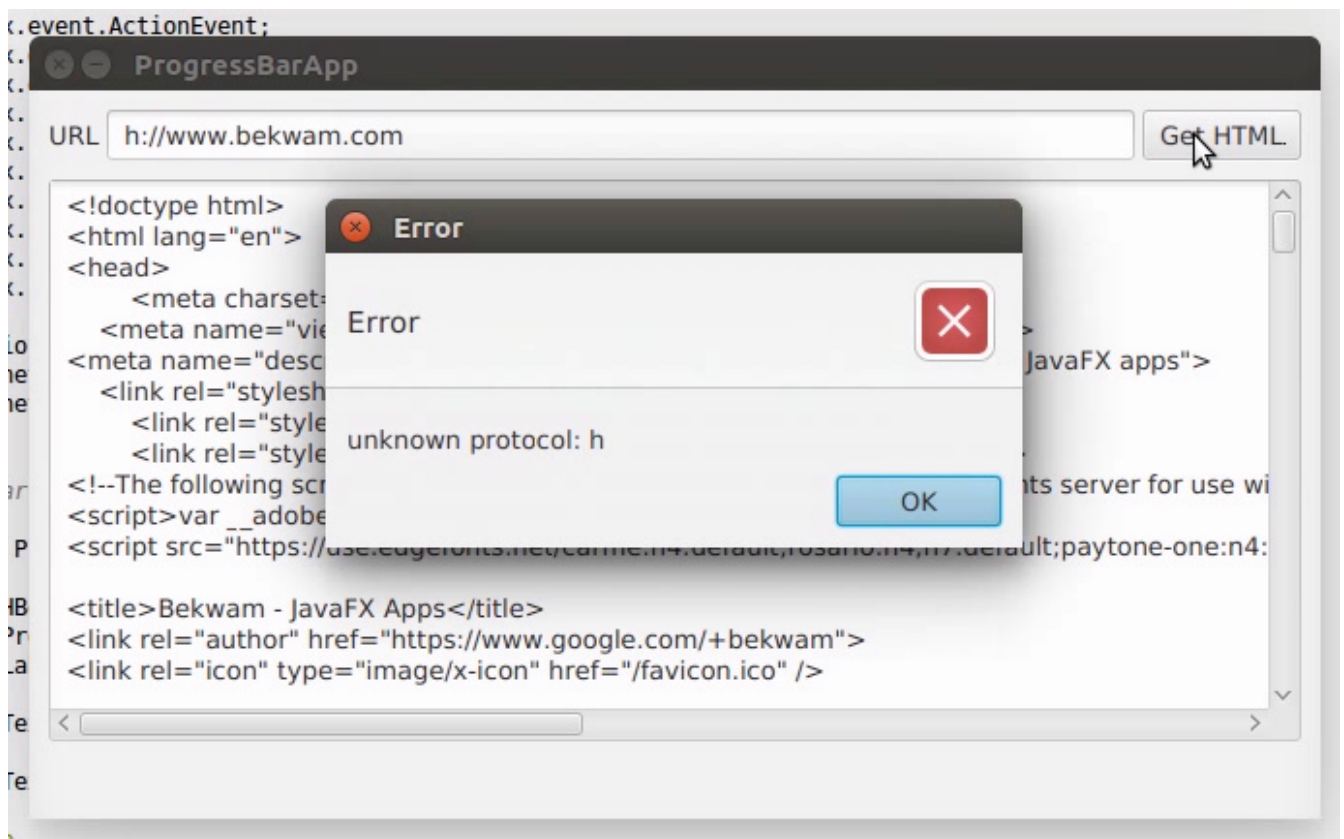


Figure 23. Screenshot of App Showing Failure

8.2.2. Code

An event handler is placed on the Get HTML Button which creates the Task. The entry point of the Task is the call() method which starts by calling updateMessage() and updateProgress(). These methods are executed on the FX Thread and will result in updates to any bound properties.

The program proceeds by issuing an HTTP GET using standard java.net classes. A String "retval" is built up from the retrieved characters. The message and progress properties are updated with more calls to updateMessage() and updateProgress(). The call() method ends with a return of the String containing the HTML text.

On a successful operation, the succeeded() callback is invoked. getValue() is a Task method that will return the value accrued in the Task (recall "retval"). The type of the value is what is provided in the generic argument, in this case "String". This could be a complex type like a domain object or a Collection. The succeeded() operation runs on the FX Thread, so the getValue() String is directly set on the TextArea.

If the operation failed, an Exception is thrown. The Exception is caught by the Task and converted to a failed() call. failed() is also FX Thread-safe and it displays an Alert.

```
String url = tfURL.getText();

Task<String> task = new Task<String>() {

    @Override
    protected String call() throws Exception {

        updateMessage("Getting HTML from " + url );
        updateProgress( 0.5d, 1.0d );

        HttpURLConnection c = null;
        InputStream is = null;
        String retval = "";

        try {

            c = (HttpURLConnection) new URL(url).openConnection();

            updateProgress( 0.6d, 1.0d );
            is = c.getInputStream();
            int ch;
            while( (ch=is.read()) != -1 ) {
                retval += (char)ch;
            }

        } finally {
            if( is != null ) {
                is.close();
            }
            if( c != null ) {
```

```

        c.disconnect();
    }
}

updateMessage("HTML retrieved");
updateProgress( 1.0d, 1.0d );

return retval;
}

@Override
protected void succeeded() {
    contents.setText( getValue() );
}

@Override
protected void failed() {
    Alert alert = new Alert(Alert.AlertType.ERROR, getException().getMessage() );
    alert.showAndWait();
}
};

```

Notice that the Task does not update the ProgressBar and status Label directly. Instead, the Task makes safe calls to `updateMessage()` and `updateProgress()`. To update the UI, JavaFX binding is used in the following statements.

```

bottomControls.visibleProperty().bind( task.runningProperty() );
pb.progressProperty().bind( task.progressProperty() );
messageLabel.textProperty().bind( task.messageProperty() );

```

`Task.runningProperty` is a boolean that can be bound to the `bottomControls HBox visibleProperty`. `Task.progressProperty` is a double that can be bound to the `ProgressBar progressProperty`. `Task.messageProperty` is a String that can be bound to the `status Label textProperty`.

To run the Task, create a Thread providing the Task as a constructor argument and invoke `start()`.

```

new Thread(task).start();

```

For any long-running operation—File IO, the Network—use a JavaFX Task to keep your application responsive. The JavaFX Task gives your application a consistent way of handling asynchronous operations and exposes several properties that can be used to eliminate boilerplate and programming logic.

8.2.3. Complete Code

The code can be tested in a single .java file.


```

public class ProgressBarApp extends Application {

    private HBox bottomControls;
    private ProgressBar pb;
    private Label messageLabel;

    private TextField tfURL;

    private TextArea contents;

    @Override
    public void start(Stage primaryStage) throws Exception {

        Parent p = createMainView();

        Scene scene = new Scene(p);

        primaryStage.setTitle("ProgressBarApp");
        primaryStage.setWidth( 667 );
        primaryStage.setHeight( 376 );
        primaryStage.setScene( scene );
        primaryStage.show();
    }

    private Parent createMainView() {

        VBox vbox = new VBox();
        vbox.setPadding( new Insets(10) );
        vbox.setSpacing( 10 );

        HBox topControls = new HBox();
        topControls.setAlignment(Pos.CENTER_LEFT);
        topControls.setSpacing( 4 );

        Label label = new Label("URL");
        tfURL = new TextField();
        HBox.setHgrow( tfURL, Priority.ALWAYS );
        Button btnGetHTML = new Button("Get HTML");
        btnGetHTML.setOnAction( this::getHTML );
        topControls.getChildren().addAll(label, tfURL, btnGetHTML);

        contents = new TextArea();
        VBox.setVgrow( contents, Priority.ALWAYS );

        bottomControls = new HBox();
        bottomControls.setVisible(false);
        bottomControls.setSpacing( 4 );
        HBox.setMargin( bottomControls, new Insets(4));

        pb = new ProgressBar();
        messageLabel = new Label("");
    }
}

```

```

        bottomControls.getChildren().addAll(pb, messageLabel);

        vbox.getChildren().addAll(topControls, contents, bottomControls);

        return vbox;
    }

    public void getHTML(ActionEvent evt) {

        String url = tfURL.getText();

        Task<String> task = new Task<String>() {

            @Override
            protected String call() throws Exception {

                updateMessage("Getting HTML from " + url );
                updateProgress( 0.5d, 1.0d );

                HttpURLConnection c = null;
                InputStream is = null;
                String retval = "";

                try {

                    c = (HttpURLConnection) new URL(url).openConnection();

                    updateProgress( 0.6d, 1.0d );
                    is = c.getInputStream();
                    int ch;
                    while( (ch=is.read()) != -1 ) {
                        retval += (char)ch;
                    }

                } finally {
                    if( is != null ) {
                        is.close();
                    }
                    if( c != null ) {
                        c.disconnect();
                    }
                }

                updateMessage("HTML retrieved");
                updateProgress( 1.0d, 1.0d );

                return retval;
            }

            @Override
            protected void succeeded() {

```

```

        contents.setText( getValue() );
    }

    @Override
    protected void failed() {
        Alert alert = new Alert(Alert.AlertType.ERROR,
getException().getMessage() );
        alert.showAndWait();
    }
};

bottomControls.visibleProperty().bind( task.runningProperty() );
pb.progressProperty().bind( task.progressProperty() );
messageLabel.textProperty().bind( task.messageProperty() );

new Thread(task).start();
}

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

```

Chapter 9. Contributing

Placeholder whilst things get built...

Chapter 10. License



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](#).