LEARNING javafx

#javafx

Table of Contents

About

Chapter 1: Getting started with javafx

Remarks

JavaFX is a software platform for creating and delivering desktop applications, as well as rich internet applications (RIAs) that can run across a wide variety of devices. JavaFX is intended to replace Swing as the standard GUI library for Java SE.

IT enables developers to design, create, test, debug, and deploy rich client applications.

The appearance of JavaFX applications can be customized using Cascading Style Sheets (CSS) for styling (see JavaFX: CSS) and (F)XML files can be used to object structures making it easy to build or develop an application (see FXML and Controllers). Scene Builder is a visual editor allowing the creation of fxml files for an UI without writing code.

Versions

Version	Release Date
JavaFX 2	2011-10-10
JavaFX 8	2014-03-18

Examples

Installation or Setup

> The JavaFX APIs are available as a fully integrated feature of the Java SE Runtime Environment (JRE) and the Java Development Kit (JDK). Because the JDK is available for all major desktop platforms (Windows, Mac OS X, and Linux), JavaFX applications compiled to JDK 7 and later also run on all the major desktop platforms. Support for ARM platforms has also been made available with JavaFX 8. JDK for ARM includes the base, graphics and controls components of JavaFX.

To install JavaFX install your chosen version of the Java Runtime environment and Java Development kit.

Features offered by JavaFX include:

1. Java APIs.
2. FXML and Scene Builder.
3. WebView.
4. Swing interoperability.
5. Built-in UI controls and CSS.
6. Modena theme.

7. 3D Graphics Features.
8. Canvas API.
9. Printing API.
10. Rich Text Support.
11. Multitouch Support.
12. Hi-DPI support.
13. Hardware-accelerated graphics pipeline.
14. High-performance media engine.
15. Self-contained application deployment model.

Hello World program

The following code creates a simple user interface containing a single `Button` that prints a `String` to the console on click.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {

    @Override
    public void start(Stage primaryStage) {
        // create a button with specified text
        Button button = new Button("Say 'Hello World'");

        // set a handler that is executed when the user activates the button
        // e.g. by clicking it or pressing enter while it's focused
        button.setOnAction(e -> {
           //Open information dialog that says hello
           Alert alert = new Alert(AlertType.INFORMATION, "Hello World!?");
           alert.showAndWait();
        });

        // the root of the scene shown in the main window
        StackPane root = new StackPane();

        // add button as child of the root
        root.getChildren().add(button);

        // create a scene specifying the root and the size
        Scene scene = new Scene(root, 500, 300);

        // add scene to the stage
        primaryStage.setScene(scene);

        // make the stage visible
        primaryStage.show();
    }

    public static void main(String[] args) {
        // launch the HelloWorld application.
        // Since this method is a member of the HelloWorld class the first
        // parameter is not required
        Application.launch(HelloWorld.class, args);
    }

}
```

The `Application` class is the entry point of every JavaFX application. Only one `Application` can be launched and this is done using

```
Application.launch(HelloWorld.class, args);
```

This creates a instance of the `Application` class passed as parameter and starts up the JavaFX platform.

The following is important for the programmer here:

1. First `launch` creates a new instance of the `Application` class (`HelloWorld` in this case). The `Application` class therefore needs a no-arg constructor.
2. `init()` is called on the `Application` instance created. In this case the default implementation from `Application` does nothing.
3. `start` is called for the `Appication` instance and the primary `Stage` (= window) is passed to the method. This method is automatically called on the JavaFX Application thread (Platform thread).
4. The application runs until the platform determines it's time to shut down. This is done when the last window is closed in this case.
5. The `stop` method is invoked on the `Application` instance. In this case the implementation from `Application` does nothing. This method is automatically called on the JavaFX Application thread (Platform thread).

In the `start` method the scene graph is constructed. In this case it contains 2 `Node`s: A `Button` and a `StackPane`.

The `Button` represents a button in the UI and the `StackPane` is a container for the `Button` that determines it's placement.

A `Scene` is created to display these `Node`s. Finally the `Scene` is added to the `Stage` which is the window that shows the whole UI.

Read Getting started with javafx online: https://riptutorial.com/javafx/topic/887/getting-started-with-javafx

Chapter 2: Animation

Examples

Animating a property with timeline

```
Button button = new Button("I'm here...");

Timeline t = new Timeline(
        new KeyFrame(Duration.seconds(0), new KeyValue(button.translateXProperty(), 0)),
        new KeyFrame(Duration.seconds(2), new KeyValue(button.translateXProperty(), 80))
);
t.setAutoReverse(true);
t.setCycleCount(Timeline.INDEFINITE);
t.play();
```

The most basic and flexible way to use animation in JavaFX is with the `Timeline` class. A timeline works by using `KeyFrame`s as known points in the animation. In this case it knows that at the start (`0 seconds`) that the `translateXProperty` needs to be zero, and at the end (`2 seconds`) that the property needs to be `80`. You can also do other things like set the animation to reverse, and how many times it should run.

Timelines can animate multiple property's at the same time:

```
Timeline t = new Timeline(
        new KeyFrame(Duration.seconds(0), new KeyValue(button.translateXProperty(), 0)),
        new KeyFrame(Duration.seconds(1), new KeyValue(button.translateYProperty(), 10)),
        new KeyFrame(Duration.seconds(2), new KeyValue(button.translateXProperty(), 80)),
        new KeyFrame(Duration.seconds(3), new KeyValue(button.translateYProperty(), 90))
);                                                                  //    ^ notice X vs Y
```

This animation will take the `Y` property from `0` (starting value of the property) to `10` one second in, and it ends at `90` at three seconds. Note that when the animation starts over that `Y` goes back to zero, even though it isn't the first value in the timeline.

Read Animation online: https://riptutorial.com/javafx/topic/5166/animation

Chapter 3: Button

Examples

Adding an action listener

Buttons fire action events when they are activated (e.g. clicked, a keybinding for the button is pressed, ...).

```
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

If you are using Java 8+, you can use lambdas for action listeners.

```
button.setOnAction((ActionEvent a) -> System.out.println("Hello, World!"));
// or
button.setOnAction(a -> System.out.println("Hello, World!"));
```

Adding a graphic to a button

Buttons can have a graphic. `graphic` can be any JavaFX node, like a `ProgressBar`

```
button.setGraphic(new ProgressBar(-1));
```

An `ImageView`

```
button.setGraphic(new ImageView("images/icon.png"));
```

Or even another button

```
button.setGraphic(new Button("Nested button"));
```

Create a Button

Creation of a `Button` is simple:

```
Button sampleButton = new Button();
```

This will create a new `Button` without any text or graphic inside.

If you want to create a `Button` with a text, simply use the constructor that takes a `String` as parameter (which sets the `textProperty` of the `Button`):

```
Button sampleButton = new Button("Click Me!");
```

If you want to create a `Button` with a graphic inside or any other `Node`, use this constructor:

```
Button sampleButton = new Button("I have an icon", new ImageView(new Image("icon.png")));
```

Default and Cancel Buttons

`Button` API provides an easy way to assign common keyboard shortcuts to buttons without the need to access accelerators' list assigned to `Scene` or explicitly listening to the key events. Namely, two convenience methods are provided: `setDefaultButton` and `setCancelButton`:

- Setting `setDefaultButton` to `true` will cause the `Button` to fire every time it receives a `KeyCode.ENTER` event.
- Setting `setCancelButton` to `true` will cause the `Button` to fire every time it receives a `KeyCode.ESCAPE` event.

The following example creates a `Scene` with two buttons that are fired when enter or escape keys are pressed, regardless whether they are focused or not.

```
FlowPane root = new FlowPane();

Button okButton = new Button("OK");
okButton.setDefaultButton(true);
okButton.setOnAction(e -> {
    System.out.println("OK clicked.");
});

Button cancelButton = new Button("Cancel");
cancelButton.setCancelButton(true);
cancelButton.setOnAction(e -> {
    System.out.println("Cancel clicked.");
});

root.getChildren().addAll(okButton, cancelButton);
Scene scene = new Scene(root);
```

The code above will not work if these `KeyEvents` are consumed by any parent `Node`:

```
scene.setOnKeyPressed(e -> {
    e.consume();
});
```

Read Button online: https://riptutorial.com/javafx/topic/5162/button

Chapter 4: Canvas

Introduction

A `Canvas` is a JavaFX `Node`, represented as a blank, rectangular area, that can display images, shapes and text. Each `Canvas` contains exactly one `GraphicsContext` object, responsible for receiving and buffering the draw calls, which, at the end, are rendered on the screen by `Canvas`.

Examples

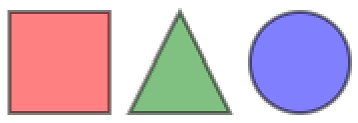
Basic shapes

`GraphicsContext` provides a set of methods to draw and fill geometric shapes. Typically, these methods need coordinates to be passed as their parameters, either directly or in a form of an array of `double` values. The coordinates are always relative to the `Canvas`, whose origin is at the top left corner.

Note: `GraphicsContext` will not draw outside of `Canvas` boundaries, i.e. trying to draw outside the `Canvas` area defined by its size and resizing it afterwards will yield no result.

The example below shows how to draw three semi-transparent filled geometric shapes outlined with a black stroke.

```
Canvas canvas = new Canvas(185, 70);
GraphicsContext gc = canvas.getGraphicsContext2D();

// Set stroke color, width, and global transparency
gc.setStroke(Color.BLACK);
gc.setLineWidth(2d);
gc.setGlobalAlpha(0.5d);

// Draw a square
gc.setFill(Color.RED);
gc.fillRect(10, 10, 50, 50);
gc.strokeRect(10, 10, 50, 50);

// Draw a triangle
gc.setFill(Color.GREEN);
gc.fillPolygon(new double[]{70, 95, 120}, new double[]{60, 10, 60}, 3);
gc.strokePolygon(new double[]{70, 95, 120}, new double[]{60, 10, 60}, 3);

// Draw a circle
gc.setFill(Color.BLUE);
gc.fillOval(130, 10, 50, 50);
gc.strokeOval(130, 10, 50, 50);
```



Read Canvas online: https://riptutorial.com/javafx/topic/8935/canvas

Chapter 5: Chart

Examples

Pie Chart

The `PieChart` class draws data in the form of circle which is divided into slices. Every slice represents a percentage (part) for a particular value. The pie chart data is wrapped in `PieChart.Data` objects. Each `PieChart.Data` object has two fields: the name of the pie slice and its corresponding value.

Constructors

To create a pie chart, we need to create the object of the `PieChart` class. Two constructors are given to our disposal. One of them creates an empty chart that will not display anything unless the data is set with `setData` method:

```
PieChart pieChart = new PieChart(); // Creates an empty pie chart
```

And the second one requires an `ObservableList` of `PieChart.Data` to be passed as a parameter.

```
ObservableList<PieChart.Data> valueList = FXCollections.observableArrayList(
            new PieChart.Data("Cats", 50),
            new PieChart.Data("Dogs", 50));
PieChart pieChart(valueList); // Creates a chart with the given data
```

Data

Values of the pie slices don't necessarily have to sum up to 100, because the slice size will be calculated in proportion to the sum of all values.

The order in which the data entries are added to the list will determine their position on the chart. By default they're laid clockwise, but this behavior can be reversed:

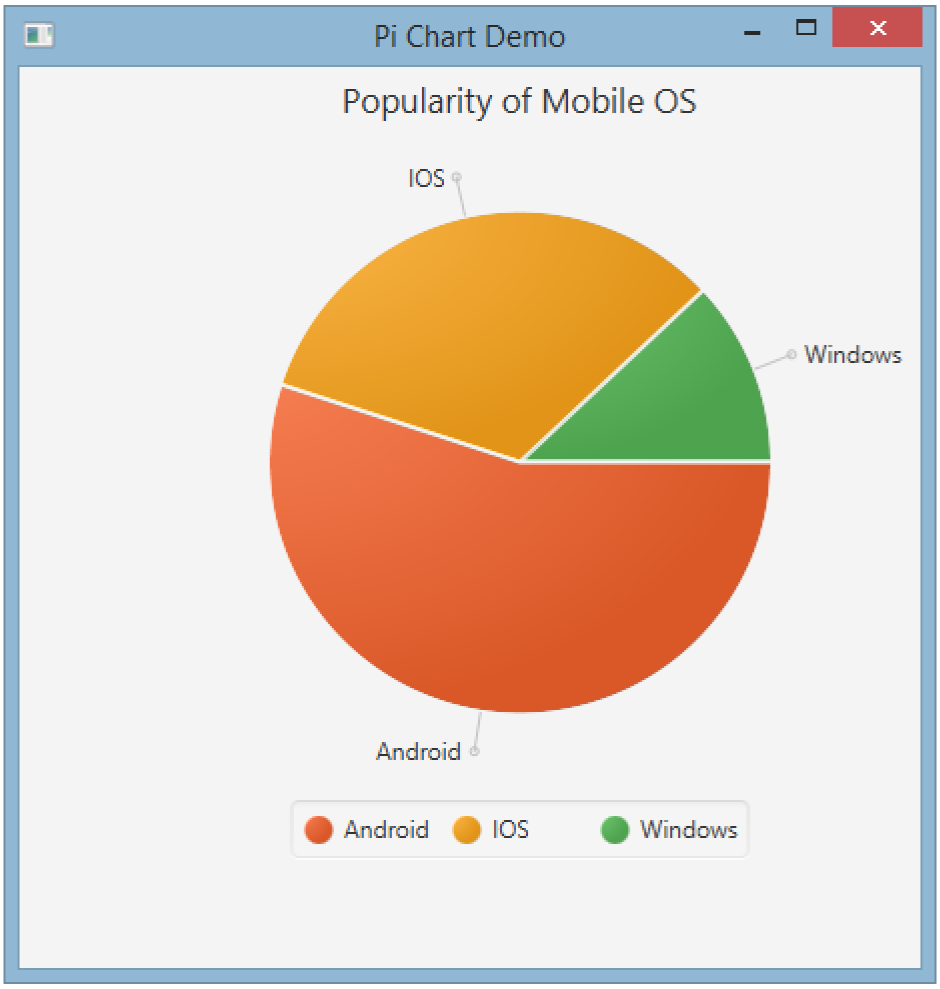
```
pieChart.setClockwise(false);
```

Example

The following example creates a simple pie chart:

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.chart.PieChart;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class Main extends Application {

@Override
public void start(Stage primaryStage) {
    Pane root = new Pane();
    ObservableList<PieChart.Data> valueList = FXCollections.observableArrayList(
            new PieChart.Data("Android", 55),
            new PieChart.Data("IOS", 33),
            new PieChart.Data("Windows", 12));
    // create a pieChart with valueList data.
    PieChart pieChart = new PieChart(valueList);
    pieChart.setTitle("Popularity of Mobile OS");
    //adding pieChart to the root.
    root.getChildren().addAll(pieChart);
    Scene scene = new Scene(root, 450, 450);

    primaryStage.setTitle("Pie Chart Demo");
    primaryStage.setScene(scene);
    primaryStage.show();
}

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

Output:



Interactive Pie Chart

By default, `PieChart` does not handle any events, but this behavior can be changed because each pie slice is a JavaFX `Node`.

In the example below we initialize the data, assign it to the chart, and then we iterate over the data set adding tooltips to each slice, so that the values, normally hidden, can be presented to the user.

```
ObservableList<PieChart.Data> valueList = FXCollections.observableArrayList(
            new PieChart.Data("Nitrogen", 7809),
            new PieChart.Data("Oxygen", 2195),
            new PieChart.Data("Other", 93));

PieChart pieChart = new PieChart(valueList);
pieChart.setTitle("Air composition");

pieChart.getData().forEach(data -> {
    String percentage = String.format("%.2f%%", (data.getPieValue() / 100));
    Tooltip toolTip = new Tooltip(percentage);
    Tooltip.install(data.getNode(), toolTip);
});
```

Line Chart

The `LineChart` class presents the data as a series of data points connected with straight lines. Each data point is wrapped in `XYChart.Data` object, and the data points are grouped in `XYChart.Series`.

Each `XYChart.Data` object has two fields, which can be accessed using `getXValue` and `getYValue`, that correspond to an x and a y value on a chart.

```
XYChart.Data data = new XYChart.Data(1,3);
System.out.println(data.getXValue()); // Will print 1
System.out.println(data.getYValue()); // Will print 3
```

Axes

Before we create a `LineChart` we need to define its axes. For example, the default, no-argument constructor of a `NumberAxis` class will create an auto-ranging axis that's ready to use and requires no further configuration.

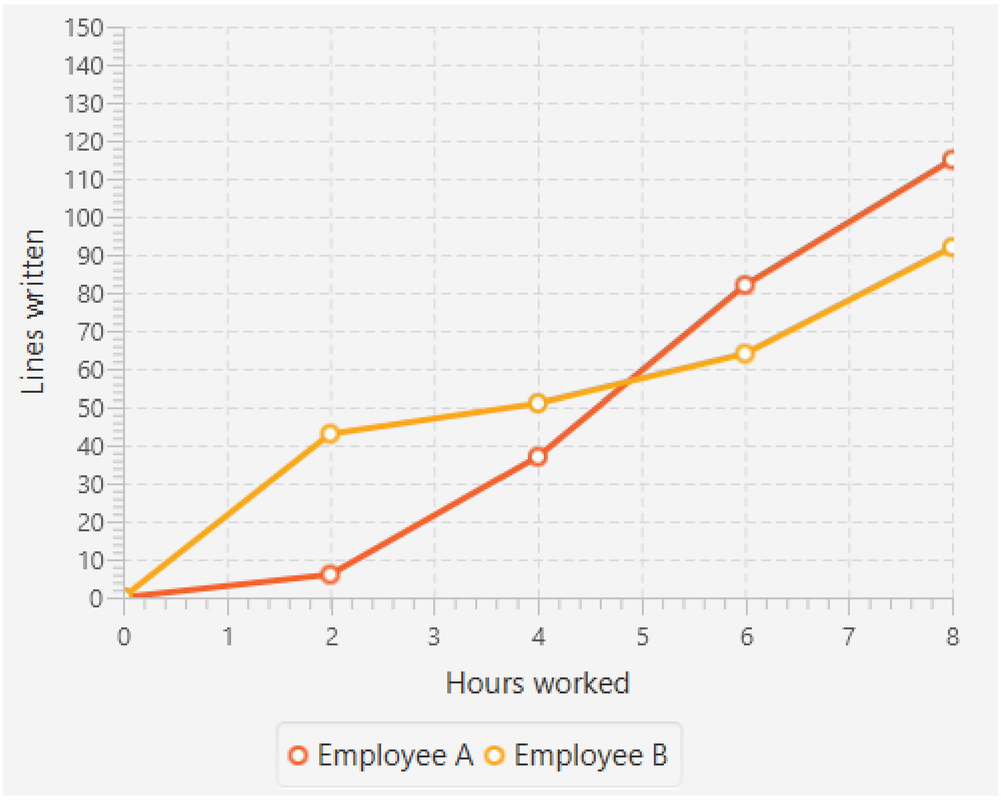
```
Axis xAxis = new NumberAxis();
```

Example

In the complete example below we create two series of data which will be displayed on the same chart. The axes' labels, ranges and tick values are explicitly defined.

```
@Override
public void start(Stage primaryStage) {
    Pane root = new Pane();

    // Create empty series
    ObservableList<XYChart.Series> seriesList = FXCollections.observableArrayList();

    // Create data set for the first employee and add it to the series
    ObservableList<XYChart.Data> aList = FXCollections.observableArrayList(
            new XYChart.Data(0, 0),
            new XYChart.Data(2, 6),
            new XYChart.Data(4, 37),
            new XYChart.Data(6, 82),
            new XYChart.Data(8, 115)
    );
    seriesList.add(new XYChart.Series("Employee A", aList));

    // Create data set for the second employee and add it to the series
    ObservableList<XYChart.Data> bList = FXCollections.observableArrayList(
            new XYChart.Data(0, 0),
            new XYChart.Data(2, 43),
            new XYChart.Data(4, 51),
            new XYChart.Data(6, 64),
            new XYChart.Data(8, 92)
    );
    seriesList.add(new XYChart.Series("Employee B", bList));
    // Create axes
    Axis xAxis = new NumberAxis("Hours worked", 0, 8, 1);
    Axis yAxis = new NumberAxis("Lines written", 0, 150, 10);

    LineChart chart = new LineChart(xAxis, yAxis, seriesList);

    root.getChildren().add(chart);

    Scene scene = new Scene(root);
    primaryStage.setScene(scene);
    primaryStage.show();
}

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

Output:



Read Chart online: https://riptutorial.com/javafx/topic/2631/chart

Chapter 6: CSS

Syntax

- NodeClass /* selector by Node's class */
- .someclass /* selector by class */
- #someId /* selector by id */
- [selector1] > [selector2] /* selector for a direct child of a node matching selector1 that matches selector2 */
- [selector1] [selector2] /* selector for a descendant of a node matching selector1 that matches selector2 */

Examples

Using CSS for styling

CSS can be applied in multiple places:

- inline (`Node.setStyle`)
- in a stylesheet
  - to a `Scene`
    - as user agent stylesheet (not demonstrated here)
    - as "normal" stylesheet for the `Scene`
  - to a `Node`

This allows to change styleable properties of `Nodes`. The following example demonstrates this:

Application class

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Region;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class StyledApplication extends Application {

    @Override
    public void start(Stage primaryStage) {

        Region region1 = new Region();
        Region region2 = new Region();
        Region region3 = new Region();
        Region region4 = new Region();
        Region region5 = new Region();
        Region region6 = new Region();

        // inline style
        region1.setStyle("-fx-background-color: yellow;");

        // set id for styling
        region2.setId("region2");

        // add class for styling
        region2.getStyleClass().add("round");
        region3.getStyleClass().add("round");

        HBox hBox = new HBox(region3, region4, region5);

        VBox vBox = new VBox(region1, hBox, region2, region6);

        Scene scene = new Scene(vBox, 500, 500);

        // add stylesheet for root
        scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());

        // add stylesheet for hBox
        hBox.getStylesheets().add(getClass().getResource("inlinestyle.css").toExternalForm());

        scene.setFill(Color.BLACK);

        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

}
```

inlinestyle.css

```
* {
    -fx-opacity: 0.5;
}

HBox {
    -fx-spacing: 10;
}

Region {
    -fx-background-color: white;
}
```

style.css

```
Region {
    width: 50;
    height: 70;

    -fx-min-width: width;
    -fx-max-width: width;

    -fx-min-height: height;
    -fx-max-height: height;

    -fx-background-color: red;
}

VBox {
    -fx-spacing: 30;
    -fx-padding: 20;
}

#region2 {
    -fx-background-color: blue;
}
```

Extending Rectangle adding new stylable properties

JavaFX 8

The following example demonstrates how to add custom properties that can be styled from css to a custom `Node`.

Here 2 `DoubleProperty`s are added to the `Rectangle` class to allow setting the `width` and `height` from CSS.

The following CSS could be used for styling the custom node:

```
 StyleableRectangle {
    -fx-fill: brown;
    -fx-width: 20;
    -fx-height: 25;
    -fx-cursor: hand;
}
```

Custom Node

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import javafx.beans.property.DoubleProperty;
import javafx.css.CssMetaData;
import javafx.css.SimpleStyleableDoubleProperty;
import javafx.css.StyleConverter;
import javafx.css.Styleable;
import javafx.css.StyleableDoubleProperty;
import javafx.css.StyleableProperty;
import javafx.scene.paint.Paint;
import javafx.scene.shape.Rectangle;

public class StyleableRectangle extends Rectangle {

    // declaration of the new properties
    private final StyleableDoubleProperty styleableWidth = new
SimpleStyleableDoubleProperty(WIDTH_META_DATA, this, "styleableWidth");
    private final StyleableDoubleProperty styleableHeight = new
SimpleStyleableDoubleProperty(HEIGHT_META_DATA, this, "styleableHeight");
```

```
    public StyleableRectangle() {
        bind();
    }

    public StyleableRectangle(double width, double height) {
        super(width, height);
        initStyleableSize();
        bind();
    }

    public StyleableRectangle(double width, double height, Paint fill) {
        super(width, height, fill);
        initStyleableSize();
        bind();
    }

    public StyleableRectangle(double x, double y, double width, double height) {
        super(x, y, width, height);
        initStyleableSize();
        bind();
    }

    private void initStyleableSize() {
        styleableWidth.set(getWidth());
        styleableHeight.set(getHeight());
    }

    private final static List<CssMetaData<? extends Styleable, ?>> CLASS_CSS_META_DATA;

    // css metadata for the width property
    // specify property name as -fx-width and
    // use converter for numbers
    private final static CssMetaData<StyleableRectangle, Number> WIDTH_META_DATA = new
CssMetaData<StyleableRectangle, Number>("-fx-width", StyleConverter.getSizeConverter()) {

        @Override
        public boolean isSettable(StyleableRectangle styleable) {
            // property can be set iff the property is not bound
            return !styleable.styleableWidth.isBound();
        }

        @Override
        public StyleableProperty<Number> getStyleableProperty(StyleableRectangle styleable) {
            // extract the property from the styleable
            return styleable.styleableWidth;
        }
    };

    // css metadata for the height property
    // specify property name as -fx-height and
    // use converter for numbers
    private final static CssMetaData<StyleableRectangle, Number> HEIGHT_META_DATA = new
CssMetaData<StyleableRectangle, Number>("-fx-height", StyleConverter.getSizeConverter()) {

        @Override
        public boolean isSettable(StyleableRectangle styleable) {
            return !styleable.styleableHeight.isBound();
        }

        @Override
        public StyleableProperty<Number> getStyleableProperty(StyleableRectangle styleable) {
            return styleable.styleableHeight;
        }
    };

    static {
        // combine already available properties in Rectangle with new properties
        List<CssMetaData<? extends Styleable, ?>> parent = Rectangle.getClassCssMetaData();
        List<CssMetaData<? extends Styleable, ?>> additional = Arrays.asList(HEIGHT_META_DATA,
WIDTH_META_DATA);

        // create arraylist with suitable capacity
        List<CssMetaData<? extends Styleable, ?>> own = new ArrayList(parent.size()+
additional.size());

        // fill list with old and new metadata
        own.addAll(parent);
        own.addAll(additional);

        // make sure the metadata list is not modifiable
        CLASS_CSS_META_DATA = Collections.unmodifiableList(own);
    }

    // make metadata available for extending the class
    public static List<CssMetaData<? extends Styleable, ?>> getClassCssMetaData() {
        return CLASS_CSS_META_DATA;
    }

    // returns a list of the css metadata for the stylable properties of the Node
    @Override
    public List<CssMetaData<? extends Styleable, ?>> getCssMetaData() {
        return CLASS_CSS_META_DATA;
    }

    private void bind() {
        this.widthProperty().bind(this.styleableWidth);
        this.heightProperty().bind(this.styleableHeight);
    }


    // -------------------------------------------------------------------------
    // ----------------------- PROPERTY METHODS --------------------------------
    // -------------------------------------------------------------------------

    public final double getStyleableHeight() {
        return this.styleableHeight.get();
    }

    public final void setStyleableHeight(double value) {
        this.styleableHeight.set(value);
    }

    public final DoubleProperty styleableHeightProperty() {
        return this.styleableHeight;
    }

    public final double getStyleableWidth() {
        return this.styleableWidth.get();
    }

    public final void setStyleableWidth(double value) {
        this.styleableWidth.set(value);
    }

    public final DoubleProperty styleableWidthProperty() {
        return this.styleableWidth;
    }

}
```

Read CSS online: https://riptutorial.com/javafx/topic/1581/css

Chapter 7: Dialogs

Remarks

Dialogs were added in JavaFX 8 update 40.

Examples

TextInputDialog

`TextInputDialog` allows the to ask the user to input a single `String`.

```
TextInputDialog dialog = new TextInputDialog("42");
dialog.setHeaderText("Input your favourite int.");
dialog.setTitle("Favourite number?");
dialog.setContentText("Your favourite int: ");

Optional<String> result = dialog.showAndWait();

String s = result.map(r -> {
    try {
        Integer n = Integer.valueOf(r);
        return MessageFormat.format("Nice! I like {0} too!", n);
    } catch (NumberFormatException ex) {
        return MessageFormat.format("Unfortunately \"{0}\" is not a int!", r);
    }
}).orElse("You really don't want to tell me, huh?");

System.out.println(s);
```

ChoiceDialog

`ChoiceDialog` allows the user to pick one item from a list of options.

```
List<String> options = new ArrayList<>();
options.add("42");
options.add("9");
options.add("467829");
options.add("Other");

ChoiceDialog<String> dialog = new ChoiceDialog<>(options.get(0), options);
dialog.setHeaderText("Choose your favourite number.");
dialog.setTitle("Favourite number?");
dialog.setContentText("Your favourite number:");

Optional<String> choice = dialog.showAndWait();

String s = choice.map(c -> "Other".equals(c) ?
              "Unfortunately your favourite number is not available!"
              : "Nice! I like " + c + " too!")
           .orElse("You really don't want to tell me, huh?");
```

```
System.out.println(s);
```

Alert

`Alert` is a simple popup that displays a set of buttons and gets an result depending on the button the user clicked:

Example

This lets the user decide, if (s)he really wants to close the primary stage:

```
@Override
public void start(Stage primaryStage) {
    Scene scene = new Scene(new Group(), 100, 100);

    primaryStage.setOnCloseRequest(evt -> {
        // allow user to decide between yes and no
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION, "Do you really want to close
this application?", ButtonType.YES, ButtonType.NO);

        // clicking X also means no
        ButtonType result = alert.showAndWait().orElse(ButtonType.NO);

        if (ButtonType.NO.equals(result)) {
            // consume event i.e. ignore close request
            evt.consume();
        }
    });
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Note that the button text is automatically adjusted depending on the `Locale`.

Custom Button text

The text displayed in a button can be customized, by creating a `ButtonType` instance yourself:

```
ButtonType answer = new ButtonType("42");
ButtonType somethingElse = new ButtonType("54");

Alert alert = new Alert(Alert.AlertType.NONE, "What do you get when you multiply six by
nine?", answer, somethingElse);
ButtonType result = alert.showAndWait().orElse(somethingElse);

Alert resultDialog = new Alert(Alert.AlertType.INFORMATION,
                               answer.equals(result) ? "Correct" : "wrong",
                               ButtonType.OK);

resultDialog.show();
```

Read Dialogs online: https://riptutorial.com/javafx/topic/3681/dialogs

Chapter 8: FXML and Controllers

Syntax

- xmlns:fx="http://javafx.com/fxml"   // namespace declaration

Examples

Example FXML

A Simple FXML document outlining an `AnchorPane` containing a button and a label node:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
xmlns:fx="http://javafx.com/fxml/1"
        fx:controller="com.example.FXMLDocumentController">
    <children>
        <Button layoutX="126" layoutY="90" text="Click Me!" onAction="#handleButtonAction"
fx:id="button" />
        <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69" fx:id="label" />
    </children>
</AnchorPane>
```

This example FXML file is associated with a controller class. The association between the FXML and the controller class, in this case, is made by specifying the class name as the value of the `fx:controller` attribute in the root element of the FXML: `fx:controller="com.example.FXMLDocumentController"`. The controller class allows for Java code to be executed in response to user actions on the UI elements defined in the FXML file:

```
package com.example ;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class FXMLDocumentController {

    @FXML
    private Label label;

    @FXML
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
        label.setText("Hello World!");
    }

    @Override
    public void initialize(URL url, ResourceBundle resources) {
        // Initialization code can go here.
        // The parameters url and resources can be omitted if they are not needed
    }

}
```

An `FXMLLoader` can be used to load the FXML file:

```
public class MyApp extends Application {

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

        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(getClass().getResource("FXMLDocument.fxml"));
        Parent root = loader.load();

        Scene scene = new Scene(root);

        stage.setScene(scene);
        stage.show();
    }

}
```

The `load` method performs several actions, and it is useful to understand the order in which they happen. In this simple example:

1. The `FXMLLoader` reads and parses the FXML file. It creates objects corresponding to the elements defined in the file, and makes note of any `fx:id` attributes defined on them.

2. Since the root element of the FXML file defined a `fx:controller` attribute, the `FXMLLoader` creates a *new instance* of the class that it specifies. By default this happens by invoking the no-argument constructor on the class specified.

3. Any elements with `fx:id` attributes defined which have fields in the controller with matching field names, and which are either `public` (not recommended) or annotated `@FXML` (recommended) are "injected" into those corresponding fields. So in this example, since there is a `Label` in the FXML file with `fx:id="label"` and a field in the controller defined as

   ```
   @FXML
   private Label label ;
   ```

   the `label` field is initialized with the `Label` instance created by the `FXMLLoader`.

4. Event handlers are registered with any elements in the FXML file with `onXXX="#..."` properties defined. These event handlers invoke the specified method in the controller class. In this example, since the `Button` has `onAction="#handleButtonAction"`, and the controller defines a

method

```
@FXML
private void handleButtonAction(ActionEvent event) { ... }
```

when an action is fired on the button (e.g. the user presses it), this method is invoked. The method must have `void` return type, and can either define a parameter matching the event type (`ActionEvent` in this example), or can define no parameters.

5. Finally, if the controller class defines an `initialize` method, this method is invoked. Notice this happens after the `@FXML` fields have been injected, so they can be safely accessed in this method and will be initialized with the instances corresponding to the elements in the FXML file. The `initialize()` method can either take no parameters, or can take a `URL` and a `ResourceBundle`. In the latter case, these parameters will be populated by the `URL` representing the location of the FXML file, and any `ResourceBundle` set on the `FXMLLoader` via `loader.setResources(...)`. Either of these can be `null` if they were not set.

Nested Controllers

There is no need to create the whole UI in a single FXML using a single controller.

The `<fx:include>` tag can be used to include one fxml file into another. The controller of the included fxml can be injected into the controller of the including file just as any other object created by the `FXMLLoader`.

This is done by adding the `fx:id` attribute to the `<fx:include>` element. This way the controller of the included fxml will be injected to the field with the name `<fx:id value>Controller`.

Examples:

fx:id value	field name for injection
foo	fooController
answer42	answer42Controller
xYz	xYzController

Sample fxmls

Counter

This is a fxml containing a `StackPane` with a `Text` node. The controller for this fxml file allows getting the current counter value as well as incrementing the counter:

counter.fxml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import javafx.scene.text.*?>
<?import javafx.scene.layout.*?>

<StackPane prefHeight="200" prefWidth="200" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="counter.CounterController">
    <children>
        <Text fx:id="counter" />
    </children>
</StackPane>
```

CounterController

```
package counter;

import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class CounterController {
    @FXML
    private Text counter;

    private int value = 0;

    public void initialize() {
        counter.setText(Integer.toString(value));
    }

    public void increment() {
        value++;
        counter.setText(Integer.toString(value));
    }

    public int getValue() {
        return value;
    }

}
```

Including fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane prefHeight="500" prefWidth="500" xmlns="http://javafx.com/javafx/8"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="counter.OuterController">
    <left>
        <Button BorderPane.alignment="CENTER" text="increment" onAction="#increment" />
    </left>
    <center>
        <!-- content from counter.fxml included here -->
        <fx:include fx:id="count" source="counter.fxml" />
    </center>
</BorderPane>
```

OuterController

The controller of the included fxml is injected to this controller. Here the handler for the `onAction` event for the `Button` is used to increment the counter.

```
package counter;

import javafx.fxml.FXML;

public class OuterController {

    // controller of counter.fxml injected here
    @FXML
    private CounterController countController;

    public void initialize() {
        // controller available in initialize method
        System.out.println("Current value: " + countController.getValue());
    }

    @FXML
    private void increment() {
        countController.increment();
    }

}
```

The fxmls can be loaded like this, assuming the code is called from a class in the same package as `outer.fxml`:

```
Parent parent = FXMLLoader.load(getClass().getResource("outer.fxml"));
```

Define Blocks and

Sometimes a element needs to be created outside of the usual object structure in the fxml.

This is where *Define Blocks* come into play:

Contents inside a `<fx:define>` element are not added to the object created for the parent element.

Every child element of the `<fx:define>` needs a `fx:id` attribute.

Objects created this way can be later referenced using the `<fx:reference>` element or by using expression binding.

The `<fx:reference>` element can be used to reference any element with a `fx:id` attribute that is handled before the `<fx:reference>` element is handled by using the same value as the `fx:id` attribute of the referenced element in the `source` attribute of the `<fx:reference>` element.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import java.lang.*?>
```

```
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>


<VBox xmlns:fx="http://javafx.com/fxml/1" prefHeight="300.0" prefWidth="300.0"
xmlns="http://javafx.com/javafx/8">
    <children>
        <fx:define>
            <String fx:value="My radio group" fx:id="text" />
        </fx:define>
        <Text>
            <text>
                <!-- reference text defined above using fx:reference -->
                <fx:reference source="text"/>
            </text>
        </Text>
        <RadioButton text="Radio 1">
            <toggleGroup>
                <ToggleGroup fx:id="group" />
            </toggleGroup>
        </RadioButton>
        <RadioButton text="Radio 2">
            <toggleGroup>
                <!-- reference ToggleGroup created for last RadioButton -->
                <fx:reference source="group"/>
            </toggleGroup>
        </RadioButton>
        <RadioButton text="Radio 3" toggleGroup="$group" />

        <!-- reference text defined above using expression binding -->
        <Text text="$text" />
    </children>
</VBox>
```

Passing data to FXML - accessing existing controller

Problem: Some data needs to be passed to a scene loaded from a fxml.

Solution

Specify a controller using the `fx:controller` attribute and get the controller instance created during the loading process from the `FXMLLoader` instance used to load the fxml.

Add methods for passing the data to the controller instance and handle the data in those methods.

FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns:fx="http://javafx.com/fxml/1" fx:controller="valuepassing.TestController">
    <children>
        <Text fx:id="target" />
    </children>
```

```
</VBox>
```

Controller

```
package valuepassing;

import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class TestController {

    @FXML
    private Text target;

    public void setData(String data) {
        target.setText(data);
    }

}
```

Code used for loading the fxml

```
String data = "Hello World!";

FXMLLoader loader = new FXMLLoader(getClass().getResource("test.fxml"));
Parent root = loader.load();
TestController controller = loader.<TestController>getController();
controller.setData(data);
```

Passing data to FXML - Specifying the controller instance

Problem: Some data needs to be passed to a scene loaded from a fxml.

Solution

Set the controller using the `FXMLLoader` instance used later to load the fxml.

Make sure the controller contains the relevant data before loading the fxml.

Note: in this case the fxml file must not contain the `fx:controller` attribute.

FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <Text fx:id="target" />
    </children>
</VBox>
```

Controller

```
import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class TestController {

    private final String data;

    public TestController(String data) {
        this.data = data;
    }

    @FXML
    private Text target;

    public void initialize() {
        // handle data once the fields are injected
        target.setText(data);
    }

}
```

Code used for loading the fxml

```
String data = "Hello World!";

FXMLLoader loader = new FXMLLoader(getClass().getResource("test.fxml"));

TestController controller = new TestController(data);
loader.setController(controller);

Parent root = loader.load();
```

Passing parameters to FXML - using a controllerFactory

Problem: Some data needs to be passed to a scene loaded from a fxml.

Solution

Specify a controller factory that is responsible for creating the controllers. Pass the data to the controller instance created by the factory.

FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns:fx="http://javafx.com/fxml/1" fx:controller="valuepassing.TestController">
    <children>
        <Text fx:id="target" />
    </children>
</VBox>
```

Controller

```
package valuepassing;

import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class TestController {

    private final String data;

    public TestController(String data) {
        this.data = data;
    }

    @FXML
    private Text target;

    public void initialize() {
        // handle data once the fields are injected
        target.setText(data);
    }

}
```

Code used for loading the fxml

String data = "Hello World!";

```
Map<Class, Callable<?>> creators = new HashMap<>();
creators.put(TestController.class, new Callable<TestController>() {

    @Override
    public TestController call() throws Exception {
        return new TestController(data);
    }

});

FXMLLoader loader = new FXMLLoader(getClass().getResource("test.fxml"));

loader.setControllerFactory(new Callback<Class<?>, Object>() {

    @Override
    public Object call(Class<?> param) {
        Callable<?> callable = creators.get(param);
        if (callable == null) {
            try {
                // default handling: use no-arg constructor
                return param.newInstance();
            } catch (InstantiationException | IllegalAccessException ex) {
                throw new IllegalStateException(ex);
            }
        } else {
            try {
                return callable.call();
            } catch (Exception ex) {
                throw new IllegalStateException(ex);
            }
        }
    }
});

Parent root = loader.load();
```

This may seem complex, but it can be useful, if the fxml should be able to decide, which controller class it needs.

Instance creation in FXML

The following class is used to demonstrate, how instances of classes can be created:

JavaFX 8

The annotation in `Person(@NamedArg("name") String name)` has to be removed, since the `@NamedArg` annotation is unavailable.

```
package fxml.sample;

import javafx.beans.NamedArg;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Person {

    public static final Person JOHN = new Person("John");

    public Person() {
        System.out.println("Person()");
    }

    public Person(@NamedArg("name") String name) {
        System.out.println("Person(String)");
        this.name.set(name);
    }

    public Person(Person person) {
        System.out.println("Person(Person)");
        this.name.set(person.getName());
    }

    private final StringProperty name = new SimpleStringProperty();

    public final String getName() {
        System.out.println("getter");
        return this.name.get();
    }

    public final void setName(String value) {
        System.out.println("setter");
        this.name.set(value);
    }

    public final StringProperty nameProperty() {
        System.out.println("property getter");
        return this.name;
    }

    public static Person valueOf(String value) {
        System.out.println("valueOf");
        return new Person(value);
    }

    public static Person createPerson() {
        System.out.println("createPerson");
        return new Person();
    }

}
```

Assume the `Person` class has already been initialized before loading the fxml.

A note on imports

In the following fxml example the imports section will be left out. However the fxml should start with

```
<?xml version="1.0" encoding="UTF-8"?>
```

followed by an imports section importing all classes used in the fxml file. Those imports are similar to non-static imports, but are added as processing instructions. **Even classes from the `java.lang` package need to be imported.**

In this case the following imports are should be added:

```
<?import java.lang.*?>
<?import fxml.sample.Person?>
```

JavaFX 8

`@NamedArg` annotated constructor

If there is a constructor where every parameter is annotated with `@NamedArg` and all values of the `@NamedArg` annotations are present in the fxml, the constructor will be used with those parameters.

```
<Person name="John"/>
```

```
<Person xmlns:fx="http://javafx.com/fxml">
    <name>
        <String fx:value="John"/>
    </name>
</Person>
```

Both result in the following console output, if loaded:

```
Person(String)
```

No args constructor

If there is no suitable `@NamedArg` annotated constructor is available, the constructor that takes no parameters will be used.

Remove the `@NamedArg` annotation from the constructor and try loading.

```
<Person name="John"/>
```

This will use the constructor without parameters.

Output:

```
Person()
setter
```

`fx:value` attribute

The `fx:value` attribute can be used to pass it's value to a `static` `valueOf` method taking a `String` parameter and returning the instance to use.

Example

```
<Person xmlns:fx="http://javafx.com/fxml" fx:value="John"/>
```

Output:

```
valueOf
Person(String)
```

`fx:factory`

The `fx:factory` attribute allows creation of objects using arbitrary `static` methods that do not take parameters.

Example

```
<Person xmlns:fx="http://javafx.com/fxml" fx:factory="createPerson">
    <name>
        <String fx:value="John"/>
    </name>
</Person>
```

Output:

```
createPerson
Person()
setter
```

`<fx:copy>`

Using `fx:copy` a copy constructor can be invoked. Specifying the `fx:id` of another The `source` attribute of the tag will invoke the copy constructor with that object as parameter.

Example:

```
<ArrayList xmlns:fx="http://javafx.com/fxml">
    <Person fx:id="p1" fx:constant="JOHN"/>
    <fx:copy source="p1"/>
</ArrayList>
```

Output

```
Person(Person)
getter
```

`fx:constant`

`fx:constant` allows getting a value from a `static final` field.

Example

```
<Person xmlns:fx="http://javafx.com/fxml" fx:constant="JOHN"/>
```

won't produce any output, since this just references `JOHN` which was created when initializing the class.

Setting Properties

There are multiple ways of adding data to a object in fxml:

`<property>` tag

A tag with the name of a property can be added as child of an element used for creating a instance. The child of this tag is assigned to the property using the setter or added to the contents of the property (readonly list/map properties).

Default property

A class can be annotated with the `@DefaultProperty` annotation. In this case elements can be directly added as child element without using a element with the name of the property.

`property="value"` attribute

Properties can be assigned using the property name as attribute name and the value as attribute value. This has the same effect as adding the following element as child of the tag:

```
<property>
    <String fx:value="value" />
</property>
```

static setters

Properties can be set using `static` setters too. These are `static` methods named `setProperty` that take the element as first parameter and the value to set as second parameter. Those methods can recide in any class and can be used using `ContainingClass.property` instead of the usual property name.

Note: Currently it seems to be neccesary to have a corresponding static getter method (i.e. a static method named `getProperty` taking the element as parameter in the same class as the static setter) for this to work unless the value type is `String`.

Type Coercion

The following mechanism is used to get a object of the correct class during assignments, e.g. to suit the parameter type of a setter method.

If the classes are assignable, then the value itself is used.

Otherwise the value is converted as follows

Target type	value used (source value `s`)
`Boolean`, `boolean`	`Boolean.valueOf(s)`
`char`, `Character`	`s.toString.charAt(0)`
other primitive	appropriate method for target type, in case the `s` is a `Number`, the

Target type	value used (source value `s`)
type or wrapper type	`valueOf(s.toString())` for the wrapper type otherwise
`BigInteger`	`BigInteger.valueOf(s.longValue())` is `s` is a `Number`, `new BigInteger(s.toString())` otherwise
`BigDecimal`	`BigDecimal.valueOf(s.doubleValue())` is `s` is a `Number`, `new BigDecimal(s.toString())` otherwise
Number	`Double.valueOf(s.toString())` if `s.toString()` contains a `.`, `Long.valueOf(s.toString())` otherwise
`Class`	`Class.forName(s.toString())` invoked using the context `ClassLoader` of the current thread without initializing the class
enum	The result of the `valueOf` method, additionally converted to an all uppercase `String` seperated by _ inserted before each uppercase letter, if `s` is a `String` that starts with a lowercase letter
other	the value returned by a `static valueOf` method in the targetType, that has a parameter matching the type of `s` or a superclass of that type

Note: This behavior isn't well-documented and could be subject to change.

Example

```
public enum Location {
    WASHINGTON_DC,
    LONDON;
}
```

```
package fxml.sample;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javafx.beans.DefaultProperty;

@DefaultProperty("items")
public class Sample {

    private Location loaction;

    public Location getLoaction() {
        return loaction;
    }

    public void setLoaction(Location loaction) {
        this.loaction = loaction;
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    int number;

    private final List<Object> items = new ArrayList<>();

    public List<Object> getItems() {
        return items;
    }

    private final Map<String, Object> map = new HashMap<>();

    public Map<String, Object> getMap() {
        return map;
    }

    private BigInteger serialNumber;

    public BigInteger getSerialNumber() {
        return serialNumber;
    }

    public void setSerialNumber(BigInteger serialNumber) {
        this.serialNumber = serialNumber;
    }

    @Override
    public String toString() {
        return "Sample{" + "loaction=" + loaction + ", number=" + number + ", items=" + items
+ ", map=" + map + ", serialNumber=" + serialNumber + '}';
    }

}
```

```
package fxml.sample;

public class Container {

    public static int getNumber(Sample sample) {
        return sample.number;
    }

    public static void setNumber(Sample sample, int number) {
        sample.number = number;
    }

    private final String value;

    private Container(String value) {
        this.value = value;
    }
    public static Container valueOf(String s) {
        return new Container(s);
    }

    @Override
    public String toString() {
        return "42" + value;
    }

}
```

Printing the result of loading the below `fxml` file yields

```
Sample{loaction=WASHINGTON_DC, number=5, items=[42a, 42b, 42c, 42d, 42e, 42f], map={answer=42,
g=9.81, hello=42A, sample=Sample{loaction=null, number=33, items=[], map={},
serialNumber=null}}, serialNumber=4299}
```

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import fxml.sample.*?>

<Sample xmlns:fx="http://javafx.com/fxml/1" Container.number="5" loaction="washingtonDc">

    <!-- set serialNumber property (type coercion) -->
    <serialNumber>
        <Container fx:value="99"/>
    </serialNumber>

    <!-- Add elements to default property-->
    <Container fx:value="a"/>
    <Container fx:value="b"/>
    <Container fx:value="c"/>
    <Container fx:value="d"/>
    <Container fx:value="e"/>
    <Container fx:value="f"/>

    <!-- fill readonly map property -->
    <map g="9.81">
        <hello>
            <Container fx:value="A"/>
        </hello>
        <answer>
            <Container fx:value=""/>
        </answer>
        <sample>
            <Sample>
                <!-- static setter-->
                <Container.number>
                    <Integer fx:value="33" />
                </Container.number>
            </Sample>
        </sample>
    </map>
</Sample>
```

Read FXML and Controllers online: https://riptutorial.com/javafx/topic/1580/fxml-and-controllers

Chapter 9: Internationalization in JavaFX

Examples

Loading Resource Bundle

JavaFX provides an easy way to internationalize your user interfaces. While creating a view from an FXML file you can provide the `FXMLLoader` with a resource bundle:

```
Locale locale = new Locale("en", "UK");
ResourceBundle bundle = ResourceBundle.getBundle("strings", locale);

Parent root = FXMLLoader.load(getClass().getClassLoader()
                                  .getResource("ui/main.fxml"), bundle);
```

This provided bundle is automatically used to translate all texts in your FXML file that start with a `%`. Lets say your properties file `strings_en_UK.properties` contains the following line:

```
ui.button.text=I'm a Button
```

If you have a button definition in your FXML like this:

```
<Button text="%ui.button.text"/>
```

It will automatically receive the translation for the key `ui.button.text`.

Controller

A Resource bundles contain locale-specific objects. You can pass the bundle to the `FXMLLoader` during its creation. The controller must implement `Initializable` interface and override `initialize(URL location, ResourceBundle resources)` method. The second parameter to this method is `ResourceBundle` which is passed from the FXMLLoader to the controller and can be used by the controller to further translate texts or modify other locale-dependant information.

```
public class MyController implements Initializable {

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        label.setText(resources.getString("country"));
    }
}
```

Switching language dynamically when the application is running

This examples shows how to build a JavaFX application, where the language can be switched dynamically while the application is running.

These are the message bundle files used in the example:

messages_en.properties:

```
window.title=Dynamic language change
button.english=English
button.german=German
label.numSwitches=Number of language switches: {0}
```

messages_de.properties:

```
window.title=Dynamischer Sprachwechsel
button.english=Englisch
button.german=Deutsch
label.numSwitches=Anzahl Sprachwechsel: {0}
```

The basic idea is to have a utility class I18N (as an alternative this might be implemented a singleton).

```
import javafx.beans.binding.Bindings;
import javafx.beans.binding.StringBinding;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.scene.control.Button;
import javafx.scene.control.Label;

import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.concurrent.Callable;

/**
 * I18N utility class..
 */
public final class I18N {

    /** the current selected Locale. */
    private static final ObjectProperty<Locale> locale;

    static {
        locale = new SimpleObjectProperty<>(getDefaultLocale());
        locale.addListener((observable, oldValue, newValue) -> Locale.setDefault(newValue));
    }

    /**
     * get the supported Locales.
     *
     * @return List of Locale objects.
     */
    public static List<Locale> getSupportedLocales() {
        return new ArrayList<>(Arrays.asList(Locale.ENGLISH, Locale.GERMAN));
    }

    /**
```

```
     * get the default locale. This is the systems default if contained in the supported
locales, english otherwise.
     *
     * @return
     */
    public static Locale getDefaultLocale() {
        Locale sysDefault = Locale.getDefault();
        return getSupportedLocales().contains(sysDefault) ? sysDefault : Locale.ENGLISH;
    }

    public static Locale getLocale() {
        return locale.get();
    }

    public static void setLocale(Locale locale) {
        localeProperty().set(locale);
        Locale.setDefault(locale);
    }

    public static ObjectProperty<Locale> localeProperty() {
        return locale;
    }

    /**
     * gets the string with the given key from the resource bundle for the current locale and
uses it as first argument
     * to MessageFormat.format, passing in the optional args and returning the result.
     *
     * @param key
     *         message key
     * @param args
     *         optional arguments for the message
     * @return localized formatted string
     */
    public static String get(final String key, final Object... args) {
        ResourceBundle bundle = ResourceBundle.getBundle("messages", getLocale());
        return MessageFormat.format(bundle.getString(key), args);
    }

    /**
     * creates a String binding to a localized String for the given message bundle key
     *
     * @param key
     *         key
     * @return String binding
     */
    public static StringBinding createStringBinding(final String key, Object... args) {
        return Bindings.createStringBinding(() -> get(key, args), locale);
    }

    /**
     * creates a String Binding to a localized String that is computed by calling the given
func
     *
     * @param func
     *         function called on every change
     * @return StringBinding
     */
    public static StringBinding createStringBinding(Callable<String> func) {
        return Bindings.createStringBinding(func, locale);
    }
    /**
     * creates a bound Label whose value is computed on language change.
     *
     * @param func
     *         the function to compute the value
     * @return Label
     */
    public static Label labelForValue(Callable<String> func) {
        Label label = new Label();
        label.textProperty().bind(createStringBinding(func));
        return label;
    }

    /**
     * creates a bound Button for the given resourcebundle key
     *
     * @param key
     *         ResourceBundle key
     * @param args
     *         optional arguments for the message
     * @return Button
     */
    public static Button buttonForKey(final String key, final Object... args) {
        Button button = new Button();
        button.textProperty().bind(createStringBinding(key, args));
        return button;
    }
}
```

This class has a static field `locale` which is a Java `Locale`object wrapped in a JavaFX `ObjectProperty`, so that bindings can be created for this property. The first methods are the standard methods to get and set a JavaFX property.

The `get(final String key, final Object... args)` is the core method that is used for the real extraction of a message from a `ResourceBundle`.

The two methods named `createStringBinding` create a `StringBinding` that is bound to the `locale`field and so the bindings will change whenever the `locale` property changes. The first one uses it's arguments to retrieve and format a message by using the `get` method mentioned above, the second one is passed in a `Callable`, which must produce the new string value.

The last two methods are methods to create JavaFX components. The first method is used to create a `Label` and uses a `Callable` for it's internal string binding. The second one creates a `Button` and uses a key value for the retrieval of the String binding.

Of course many more different objects could be created like `MenuItem` or `ToolTip` but these two should be enough for an example.

This code shows how this class is used within the application:

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

import java.util.Locale;

/**
 * Sample application showing dynamic language switching,
 */
public class I18nApplication extends Application {

    /** number of language switches. */
    private Integer numSwitches = 0;

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

        primaryStage.titleProperty().bind(I18N.createStringBinding("window.title"));

        // create content
        BorderPane content = new BorderPane();

        // at the top two buttons
        HBox hbox = new HBox();
        hbox.setPadding(new Insets(5, 5, 5, 5));
        hbox.setSpacing(5);

        Button buttonEnglish = I18N.buttonForKey("button.english");
        buttonEnglish.setOnAction((evt) -> switchLanguage(Locale.ENGLISH));
        hbox.getChildren().add(buttonEnglish);

        Button buttonGerman = I18N.buttonForKey("button.german");
        buttonGerman.setOnAction((evt) -> switchLanguage(Locale.GERMAN));
        hbox.getChildren().add(buttonGerman);

        content.setTop(hbox);

        // a label to display the number of changes, recalculating the text on every change
        final Label label = I18N.labelForValue(() -> I18N.get("label.numSwitches",
numSwitches));
        content.setBottom(label);

        primaryStage.setScene(new Scene(content, 400, 200));
        primaryStage.show();
    }

    /**
     * sets the given Locale in the I18N class and keeps count of the number of switches.
     *
     * @param locale
     *         the new local to set
     */
    private void switchLanguage(Locale locale) {
        numSwitches++;
        I18N.setLocale(locale);
    }
}
```

The application shows three different ways of using the `StringBinding` created by the `I18N`class:

1. the window title is bound by directly using a `StringBinding`.
2. the buttons use the helper method with the message keys
3. the label uses the helper method with a `Callable`. This `Callable` uses the `I18N.get()` method to get a formatted translated string containing the actual count of switches.

On clicking a button, the counter is increased and the `I18N`s locale property is set, which in turn triggers the string bindings changing and so setting the UI's string to new values.

Read Internationalization in JavaFX online: https://riptutorial.com/javafx/topic/5434/internationalization-in-javafx

Chapter 10: JavaFX bindings

Examples

Simple property binding

JavaFX has a binding API, which provides ways of binding one property to the other. This means that whenever one property's value is changed, the value of the bound property is updated automatically. An example of simple binding:

```
SimpleIntegerProperty first =new SimpleIntegerProperty(5); //create a property with value=5
SimpleIntegerProperty second=new SimpleIntegerProperty();

public void test()
{
    System.out.println(second.get()); // '0'
    second.bind(first);               //bind second property to first
    System.out.println(second.get()); // '5'
    first.set(16);                    //set first property's value
    System.out.println(second.get()); // '16' - the value was automatically updated
}
```

You can also bind a primitive property with applying an addition, subtraction, division, etc:

```
public void test2()
{
        second.bind(first.add(100));
        System.out.println(second.get()); //'105'
        second.bind(first.subtract(50));
        System.out.println(second.get()); //'-45'
}
```

Any Object can be put into SimpleObjectProperty:

```
SimpleObjectProperty<Color> color=new SimpleObjectProperty<>(Color.web("45f3d1"));
```

It is possible to create bidirectional bindings. In this case, properties depend on each other.

```
public void test3()
{
        second.bindBidirectional(first);
        System.out.println(second.get()+" "+first.get());
        second.set(1000);
        System.out.println(second.get()+" "+first.get()); //both are '1000'
}
```

Read JavaFX bindings online: https://riptutorial.com/javafx/topic/7014/javafx-bindings

Chapter 11: Layouts

Examples

StackPane

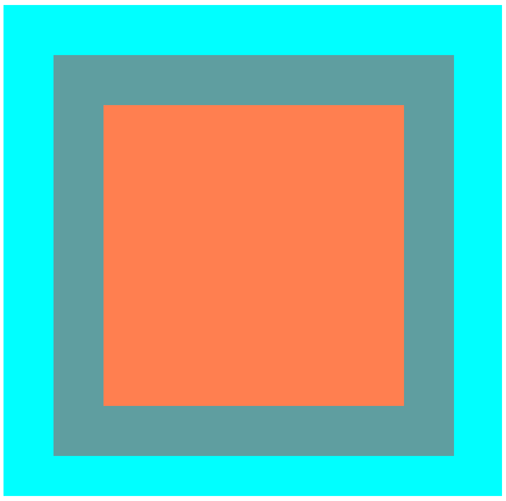
`StackPane` lays out its children in a back-to-front stack.

The z-order of the children is defined by the order of the children list (accessible by calling `getChildren`): the 0th child being the bottom and last child on top of the stack.

The stackpane attempts to resize each child to fill its own content area. In the case if a child cannot be resized to fill the area of the `StackPane` (either because it was not resizable or its max size prevented it) then it will be aligned within the area using the `alignmentProperty` of the stackpane, which defaults to `Pos.CENTER`.

Example

```
// Create a StackPane
StackPane pane = new StackPane();

// Create three squares
Rectangle rectBottom = new Rectangle(250, 250);
rectBottom.setFill(Color.AQUA);
Rectangle rectMiddle = new Rectangle(200, 200);
rectMiddle.setFill(Color.CADETBLUE);
Rectangle rectUpper = new Rectangle(150, 150);
rectUpper.setFill(Color.CORAL);

// Place them on top of each other
pane.getChildren().addAll(rectBottom, rectMiddle, rectUpper);
```



HBox and VBox

The `HBox` and `VBox` layouts are very similar, both lay out their children in a single line.

Common characteristics

If an `HBox` or a `VBox` have a border and/or padding set, then the contents will be layed out within those insets.

They lay out each managed child regardless of the child's visible property value; unmanaged children are ignored.

The alignment of the content is controlled by the alignment property, which defaults to **`Pos.TOP_LEFT`**
.

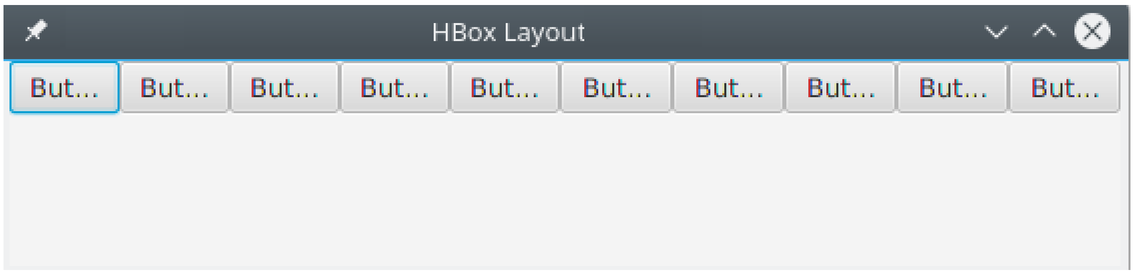
HBox

`HBox` lays out its children in a single horizontal row from left to right.

`HBox` will resize children (if resizable) **to their preferred width**s and uses its fillHeight property to determine whether to resize their heights to fill its own height or keep their heights to their preferred (fillHeight defaults to true).

Creating a HBox

```
// HBox example
HBox row = new HBox();
Label first = new Label("First");
Label second = new Label("Second");
row.getChildren().addAll(first, second);
```



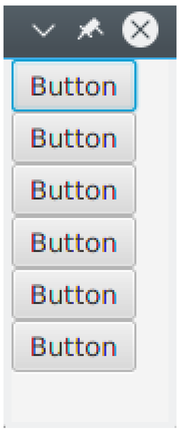
VBox

`VBox` lays out its children in a single vertical column from top to bottom.

`VBox` will resize children (if resizable) **to their preferred heights** and uses its fillWidth property to determine whether to resize their widths to fill its own width or keep their widths to their preferred (fillWidth defaults to true).

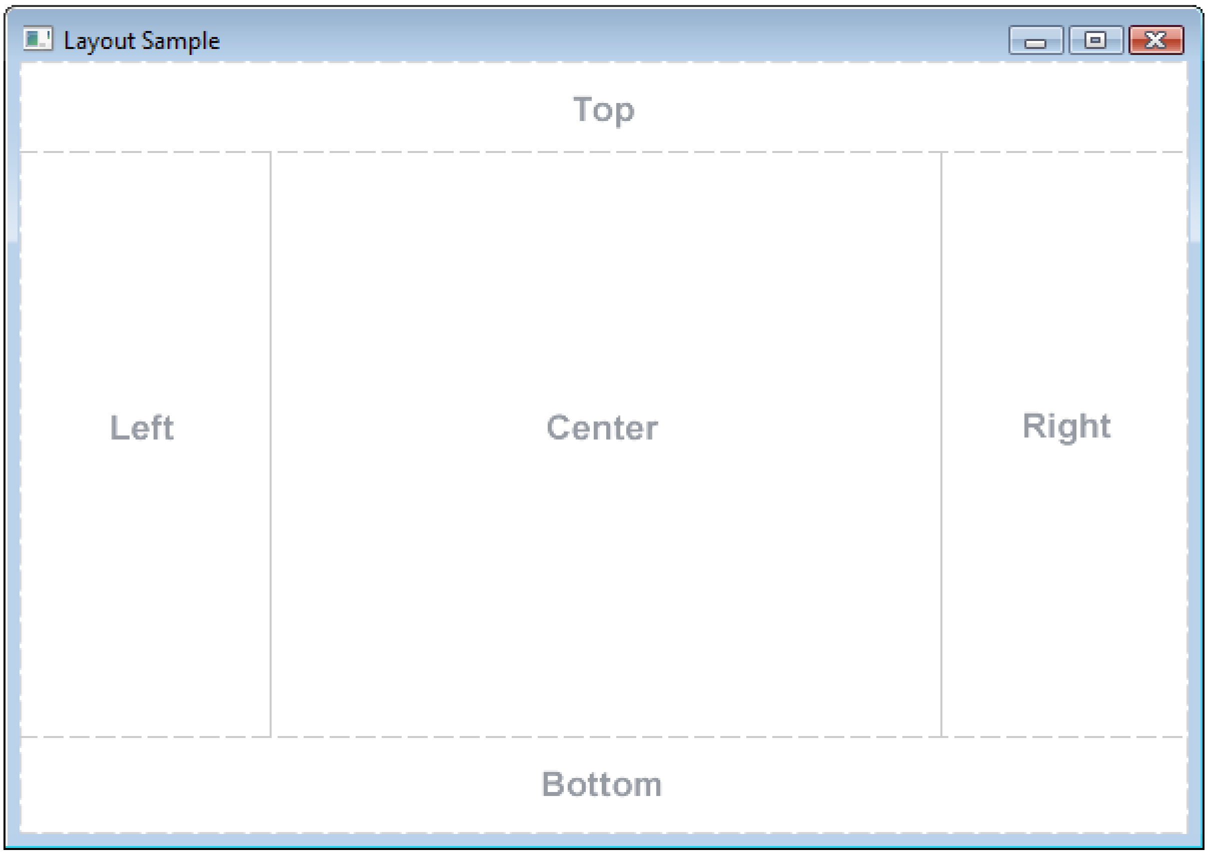
Creating a VBox

```
// VBox example
VBox column = new VBox();
Label upper = new Label("Upper");
Label lower = new Label("Lower");
column.getChildren().addAll(upper, lower);
```



BorderPane

The `BorderPane` is separated into five different areas.



The border areas (`Top`, `Right`, `Bottom`, `Left`) have preferred sized based on their content. By default they will only take what they need, while the `Center` area will take any remaining space. When the border areas are empty, they do not take up any space.

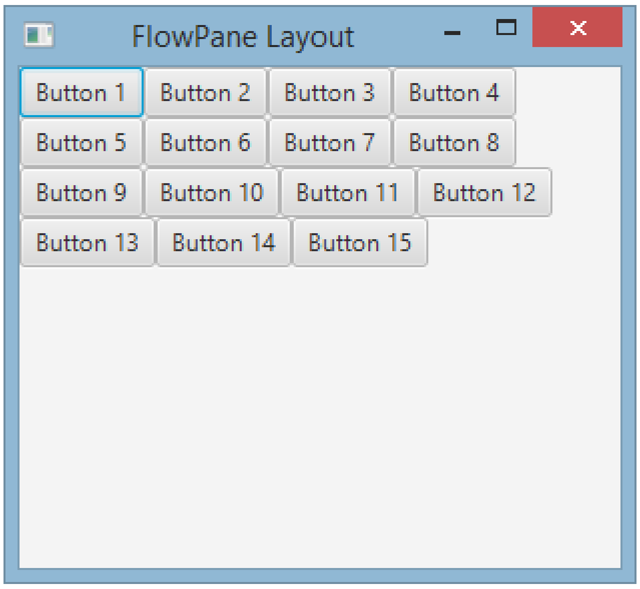
Each area can contain only one element. It can be added using the methods `setTop(Node)`, `setRight(Node)`, `setBottom(Node)`, `setLeft(Node)`, `setCenter(Node)`. You can use other layouts to put more than one element into a single area.

```
//BorderPane example
BorderPane pane = new BorderPane();

Label top = new Label("Top");
Label right = new Label("Right");

HBox bottom = new HBox();
bottom.getChildren().addAll(new Label("First"), new Label("Second"));

VBox left = new VBox();
left.getChildren().addAll(new Label("Upper"), new Label("Lower"));

StackPane center = new StackPane();
center.getChildren().addAll(new Label("Lorem"), new Label("ipsum"));

pane.setTop(top);        //The text "Top"
pane.setRight(right);    //The text "Right"
pane.setBottom(bottom);  //Row of two texts
pane.setLeft(left);      //Column of two texts
pane.setCenter(center);  //Two texts on each other
```

FlowPane

`FlowPane` lays out nodes in rows or columns based on the available horizontal or vertical space available. It wraps nodes to the next line when the horizontal space is less than the total of all the nodes' widths; it wraps nodes to the next column when the vertical space is less than the total of all the nodes' heights. This example illustrates the default horizontal layout:

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception{
        FlowPane root = new FlowPane();
        for (int i=1; i<=15; i++) {
            Button b1=new Button("Button "+String.valueOf(i));
            root.getChildren().add(b1); //for adding button to root
        }
        Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("FlowPane Layout");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```



Default `FlowPane` constructor:

```
FlowPane root = new FlowPane();
```

Additional `FlowPane` constructors:

```
FlowPane() //Creates a horizontal FlowPane layout with hgap/vgap = 0 by default.
FlowPane(double hgap, double gap) //Creates a horizontal FlowPane layout with the specified
hgap/vgap.
FlowPane(double hgap, double vgap, Node... children) //Creates a horizontal FlowPane layout
with the specified hgap/vgap.
FlowPane(Node... children) //Creates a horizontal FlowPane layout with hgap/vgap = 0.
FlowPane(Orientation orientation) //Creates a FlowPane layout with the specified orientation
and hgap/vgap = 0.
FlowPane(Orientation orientation, double hgap, double gap) //Creates a FlowPane layout with
the specified orientation and hgap/vgap.
FlowPane(Orientation orientation, double hgap, double vgap, Node... children) //Creates a
FlowPane layout with the specified orientation and hgap/vgap.
FlowPane(Orientation orientation, Node... children) //Creates a FlowPane layout with the
specified orientation and hgap/vgap = 0.
```

Adding nodes to the layout uses the `add()` or `addAll()` methods of the parent `Pane`:

```
Button btn = new Button("Demo Button");
root.getChildren().add(btn);
root.getChildren().addAll(…);
```

By default, a `FlowPane` lays out child nodes from left to right. To change the flow alignment, call the `setAlignment()` method by passing in an enumerated value of type `Pos`.

Some commonly used flow alignments:

```
root.setAlignment(Pos.TOP_RIGHT);     //for top right
root.setAlignment(Pos.TOP_CENTER);    //for top Center
root.setAlignment(Pos.CENTER);        //for Center
root.setAlignment(Pos.BOTTOM_RIGHT);  //for bottom right
```

GridPane

`GridPane` lays out its children within a flexible grid of rows and columns.

Children of the GridPane

A child may be placed anywhere within the `GridPane` and may span multiple rows/columns (default span is 1) and its placement within the grid is defined by it's layout constraints:

Constraint	Description
columnIndex	column where child's layout area starts.
rowIndex	row where child's layout area starts.
columnSpan	the number of columns the child's layout area spans horizontally.
rowSpan	the number of rows the child's layout area spans vertically.

The total number of rows/columns does not need to be specified up front as the gridpane will automatically expand/contract the grid to accommodate the content.

Adding children to the GridPane

In order to add new `Node`s to a `GridPane` the **layout constraints** on the children should be set using the static method of `GridPane` class, then those children can be added to a `GridPane` instance.

```
GridPane gridPane = new GridPane();

// Set the constraints: first row and first column
Label label = new Label("Example");
GridPane.setRowIndex(label, 0);
GridPane.setColumnIndex(label, 0);
// Add the child to the grid
gridpane.getChildren().add(label);
```

`GridPane` provides convenient methods to combine these steps:

```
gridPane.add(new Button("Press me!"), 1, 0); // column=1 row=0
```

The `GridPane` class also provides static setter methods to set the **row- and columnspan** of child elements:

```
Label labelLong = new Label("Its a long text that should span several rows");
GridPane.setColumnSpan(labelLong, 2);
gridPane.add(labelLong, 0, 1);  // column=0 row=1
```

Size of Columns and Rows

By default, rows and columns will be sized to fit their content. In case of the need of the **explicit control of row and column sizes**, `RowConstraints` and `ColumnConstraints` instances can be added to the `GridPane`. Adding these two constraints will resize the example above to have the first column 100 pixels, the second column 200 pixels long.

```
gridPane.getColumnConstraints().add(new ColumnConstraints(100));
gridPane.getColumnConstraints().add(new ColumnConstraints(200));
```

By default the `GridPane` will resize rows/columns to their preferred sizes even if the gridpane is resized larger than its preferred size. To support **dynamic column/row sizes**, both contstaints class provides three property: min size, max size and preferred size.

Additionally `ColumnConstraints` provides `setHGrow` and `RowConstraints` provides `setVGrow` methods to **affect the priority of the growing and shrinking**. The three pre-defined priorities are:

- **Priority.ALWAYS**: Always try to grow (or shrink), sharing the increase (or decrease) in space with other layout areas that have a grow (or shrink) of ALWAYS
- **Priority.SOMETIMES**: If there are no other layout areas with grow (or shrink) set to ALWAYS or those layout areas didn't absorb all of the increased (or decreased) space, then will share the increase (or decrease) in space with other layout area's of SOMETIMES.
- **Priority.NEVER**: Layout area will never grow (or shrink) when there is an increase (or decrease) in space available in the region.

```
ColumnConstraints column1 = new ColumnConstraints(100, 100, 300);
column1.setHgrow(Priority.ALWAYS);
```

The column defined above have a minimal size of 100 pixels and it will always try to grow until it reaches its maximal 300 pixel width.

It is also possible to define **percentage sizing** for rows and columns. The following example defines a `GridPane` where the first column fills 40% of the gridpane's width, the second one fills the 60%.

```
GridPane gridpane = new GridPane();
ColumnConstraints column1 = new ColumnConstraints();
column1.setPercentWidth(40);
ColumnConstraints column2 = new ColumnConstraints();
column2.setPercentWidth(60);
gridpane.getColumnConstraints().addAll(column1, column2);
```

Alignment of elements inside the grid cells

The alignment of `Node`s can be defined by using the `setHalignment` (horizontal) method of `ColumnConstraints` class and `setValignment` (vertical) method of `RowConstraints` class.

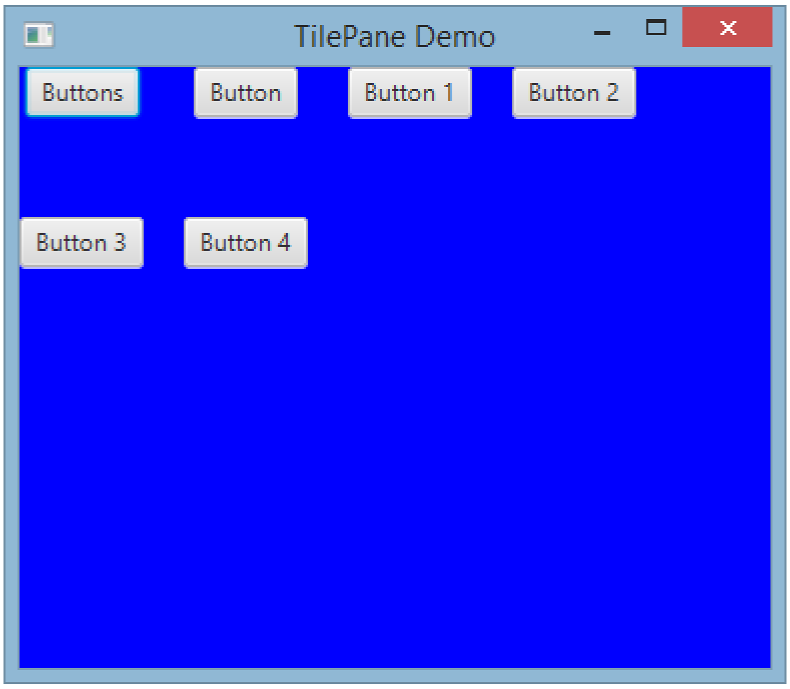
```
ColumnConstraints column1 = new ColumnConstraints();
column1.setHalignment(HPos.RIGHT);

RowConstraints row1 = new RowConstraints();
row1.setValignment(VPos.CENTER);
```

TilePane

The tile pane layout is similar to the FlowPane layout. TilePane places all of the nodes in a grid in which each cell, or tile, is the same size. It arranges nodes in neat rows and columns, either horizontally or vertically.

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

    public class Main extends Application {

        @Override
        public void start(Stage primaryStage) {
            primaryStage.setTitle("TilePane Demo");
            double width = 400;
            double height = 300;
            TilePane root = new TilePane();
            root.setStyle("-fx-background-color:blue");
            // to set horizontal and vertical gap
            root.setHgap(20);
            root.setVgap(50);
            Button bl = new Button("Buttons");
            root.getChildren().add(bl);
            Button btn = new Button("Button");
            root.getChildren().add(btn);
            Button btn1 = new Button("Button 1");
            root.getChildren().add(btn1);
            Button btn2 = new Button("Button 2");
            root.getChildren().add(btn2);
            Button btn3 = new Button("Button 3");
            root.getChildren().add(btn3);
            Button btn4 = new Button("Button 4");
            root.getChildren().add(btn4);

            Scene scene = new Scene(root, width, height);
            primaryStage.setScene(scene);
            primaryStage.show();
        }


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

output



To create Tilepane

```
TilePane root = new TilePane();
```

setHgap() And setVgap() method is used to make gap between column and column. we can also set the columns for the layout by using

```
int columnCount = 2;
root.setPrefColumns(columnCount);
```

AnchorPane

`AnchorPane` a is a layout that allows placing the content at a specific distance from it's sides.

There are 4 methods for setting and 4 methods for getting the distances in `AnchorPane`. The first parameter of these methods is the child `Node`. The second parameter of the setters is the `Double` value to use. This value can be `null` indicating no constraint for the given side.

setter method	getter method
setBottomAnchor	getBottomAnchor
setLeftAnchor	getLeftAnchor
setRightAnchor	getRightAnchor
setTopAnchor	getTopAnchor

In the following example places nodes at specified distances from the sides.

The `center` region is also resized to keep the specified distances from the sides. Observe the behaviour when the window is resized.

```
public static void setBackgroundColor(Region region, Color color) {
    // change to 50% opacity
    color = color.deriveColor(0, 1, 1, 0.5);
    region.setBackground(new Background(new BackgroundFill(color, CornerRadii.EMPTY,
Insets.EMPTY)));
}

@Override
public void start(Stage primaryStage) {
    Region right = new Region();
    Region top = new Region();
    Region left = new Region();
    Region bottom = new Region();
    Region center = new Region();

    right.setPrefSize(50, 150);
    top.setPrefSize(150, 50);
    left.setPrefSize(50, 150);
    bottom.setPrefSize(150, 50);

    // fill with different half-transparent colors
    setBackgroundColor(right, Color.RED);
    setBackgroundColor(left, Color.LIME);
    setBackgroundColor(top, Color.BLUE);
    setBackgroundColor(bottom, Color.YELLOW);
    setBackgroundColor(center, Color.BLACK);

    // set distances to sides
    AnchorPane.setBottomAnchor(bottom, 50d);
    AnchorPane.setTopAnchor(top, 50d);
    AnchorPane.setLeftAnchor(left, 50d);
    AnchorPane.setRightAnchor(right, 50d);

    AnchorPane.setBottomAnchor(center, 50d);
    AnchorPane.setTopAnchor(center, 50d);
    AnchorPane.setLeftAnchor(center, 50d);
    AnchorPane.setRightAnchor(center, 50d);

    // create AnchorPane with specified children
    AnchorPane anchorPane = new AnchorPane(left, top, right, bottom, center);

    Scene scene = new Scene(anchorPane, 200, 200);

    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Read Layouts online: https://riptutorial.com/javafx/topic/2121/layouts

Chapter 12: Pagination

Examples

Creating a pagination

Paginations in JavaFX use a callback to get the pages used in the animation.

```
Pagination p = new Pagination();
p.setPageFactory(param -> new Button(param.toString()));
```

This creates an infinite list of buttons numbed `0..` since the zero arg constructor creates an infinite pagination. `setPageFactory` takes a callback that takes an int, and returns the node that we want at that index.

Auto advance

```
Pagination p = new Pagination(10);

Timeline fiveSecondsWonder = new Timeline(new KeyFrame(Duration.seconds(5), event -> {
    int pos = (p.getCurrentPageIndex()+1) % p.getPageCount();
    p.setCurrentPageIndex(pos);
}));
fiveSecondsWonder.setCycleCount(Timeline.INDEFINITE);
fiveSecondsWonder.play();

stage.setScene(new Scene(p));
stage.show();
```

This advances the pagination every 5 seconds.

How it works

```
Pagination p = new Pagination(10);

Timeline fiveSecondsWonder = new Timeline(new KeyFrame(Duration.seconds(5), event -> {
```

`fiveSecondsWonder` is a timeline that fires an event every time it finishes a cycle. In this case the cycle time is 5 seconds.

```
    int pos = (p.getCurrentPageIndex()+1) % p.getPageCount();
    p.setCurrentPageIndex(pos);
```

Tick the pagination.

```
}));
fiveSecondsWonder.setCycleCount(Timeline.INDEFINITE);
```

Set the timeline to run forever.

```
fiveSecondsWonder.play();
```

Create a pagination of images

```
ArrayList<String> images = new ArrayList<>();
images.add("some\\cool\\image");
images.add("some\\other\\cool\\image");
images.add("some\\cooler\\image");

Pagination p = new Pagination(3);
p.setPageFactory(n -> new ImageView(images.get(n)));
```

Note that the paths must be urls, not filesystem paths.

How it works

```
p.setPageFactory(n -> new ImageView(images.get(n)));
```

Everything else is just fluff, this is where the real work is happening. `setPageFactory` takes a callback that takes an int, and returns the node that we want at that index. The first page maps to the first item in the list, the second to the second item in the list and so on.

Read Pagination online: https://riptutorial.com/javafx/topic/5165/pagination

Chapter 13: Printing

Examples

Basic printing

```
PrinterJob pJ = PrinterJob.createPrinterJob();

if (pJ != null) {
    boolean success = pJ.printPage(some-node);
    if (success) {
        pJ.endJob();
    }
}
```

This prints to the default printer without showing any dialog to the user. To use a printer other than the default you can use the `PrinterJob#createPrinterJob(Printer)` to set the current printer. You can use this to view all printers on your system:

```
System.out.println(Printer.getAllPrinters());
```

Printing with system dialog

```
PrinterJob pJ = PrinterJob.createPrinterJob();

if (pJ != null) {
    boolean success = pJ.showPrintDialog(primaryStage);// this is the important line
    if (success) {
        pJ.endJob();
    }
}
```

Read Printing online: https://riptutorial.com/javafx/topic/5157/printing

Chapter 14: Properties & Observable

Remarks

Properties are observable and listeners can be added to them. They are consistently used for properties of `Node`s.

Examples

Types of properties and naming

Standard properties

Depending on the type of the property, there are up to 3 methods for a single property. Let `<property>` denote the name of a property and `<Property>` the name of the property with an uppercase first letter. And let `T` be the type of the property; for primitive wrappers we use the primitive type here, e.g. `String` for `StringProperty` and `double` for `ReadOnlyDoubleProperty`.

Method name	Parameters	Return type	Purpose
`<property>Property`	`()`	The property itself, e.g. `DoubleProperty`, `ReadOnlyStringProperty`, `ObjectProperty<VPos>`	return the property itself for adding listeners / binding
`get<Property>`	`()`	`T`	return the value wrapped in the property
`set<Property>`	`(T)`	`void`	set the value of the property

Note that the setter does not exist for readonly properties.

Readonly list properties

Readonly list properties are properties that provide only a getter method. The type of such a property is `ObservableList`, preferably with a type agrument specified. The value of this property never changes; the content of the `ObservableList` may be changed instead.

Readonly map properties

Similar to readonly list properties readonly map properties only provide a getter and the content may be modified instead of the property value. The getter returns a `ObservableMap`.

StringProperty example

The following example shows the declaration of a property (`StringProperty` in this case) and demonstrates how to add a `ChangeListener` to it.

```
import java.text.MessageFormat;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;

public class Person {

    private final StringProperty name = new SimpleStringProperty();

    public final String getName() {
        return this.name.get();
    }

    public final void setName(String value) {
        this.name.set(value);
    }

    public final StringProperty nameProperty() {
        return this.name;
    }

    public static void main(String[] args) {
        Person person = new Person();
        person.nameProperty().addListener(new ChangeListener<String>() {

            @Override
            public void changed(ObservableValue<? extends String> observable, String oldValue,
String newValue) {
                System.out.println(MessageFormat.format("The name changed from \"{0}\" to
\"{1}\"", oldValue, newValue));
            }

        });

        person.setName("Anakin Skywalker");
        person.setName("Darth Vader");
    }

}
```

ReadOnlyIntegerProperty example

This example shows how to use a readonly wrapper property to create a property that cannot be written to. In this case `cost` and `price` can be modified, but `profit` will always be `price - cost`.

```
import java.text.MessageFormat;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.ReadOnlyIntegerProperty;
import javafx.beans.property.ReadOnlyIntegerWrapper;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;

public class Product {

    private final IntegerProperty price = new SimpleIntegerProperty();
    private final IntegerProperty cost = new SimpleIntegerProperty();
    private final ReadOnlyIntegerWrapper profit = new ReadOnlyIntegerWrapper();

    public Product() {
        // the property itself can be written to
        profit.bind(price.subtract(cost));
    }

    public final int getCost() {
        return this.cost.get();
    }

    public final void setCost(int value) {
        this.cost.set(value);
    }

    public final IntegerProperty costProperty() {
        return this.cost;
    }

    public final int getPrice() {
        return this.price.get();
    }

    public final void setPrice(int value) {
        this.price.set(value);
    }

    public final IntegerProperty priceProperty() {
        return this.price;
    }

    public final int getProfit() {
        return this.profit.get();
    }

    public final ReadOnlyIntegerProperty profitProperty() {
        // return a readonly view of the property
        return this.profit.getReadOnlyProperty();
    }

    public static void main(String[] args) {
        Product product = new Product();
        product.profitProperty().addListener(new ChangeListener<Number>() {

            @Override
            public void changed(ObservableValue<? extends Number> observable, Number oldValue,
Number newValue) {
                System.out.println(MessageFormat.format("The profit changed from {0}$ to
{1}$", oldValue, newValue));
            }

        });
        product.setCost(40);
        product.setPrice(50);
        product.setCost(20);
        product.setPrice(30);
    }

}
```

Read Properties & Observable online: https://riptutorial.com/javafx/topic/4436/properties---observable

Chapter 15: Radio Button

Examples

Creating Radio Buttons

Radio Buttons allow you to let the user choose one element of those given. There are two ways to declare a `RadioButton` with a text besides it. Either by using the default constructor `RadioButton()` and setting the text with the `setText(String)` method or by using the other constructor `RadioButton(String)`.

```
RadioButton radioButton1 = new RadioButton();
radioButton1.setText("Select me!");
RadioButton radioButton2= new RadioButton("Or me!");
```

As `RadioButton` is an extension of `Labeled` there can also be an `Image` specified to the `RadioButton`. After creating the `RadioButton` with one of the constructors simply add the `Image` with the `setGraphic(ImageView)` method like here:

```
Image image = new Image("ok.jpg");
RadioButton radioButton = new RadioButton("Agree");
radioButton.setGraphic(new ImageView(image));
```

Use Groups on Radio Buttons

A `ToggleGroup` is used to manage the `RadioButton`s so that just one in each group can be selected at each time.

Create a simple `ToggleGroup` like following:

```
ToggleGroup group = new ToggleGroup();
```

After creating a `Togglegroup` it can be assigned to the `RadioButton`s by using `setToggleGroup(ToggleGroup)`. Use `setSelected(Boolean)` to pre-select one of the `RadioButton`s.

```
RadioButton radioButton1 = new RadioButton("stackoverlow is awesome! :)");
radioButton1.setToggleGroup(group);
radioButton1.setSelected(true);

RadioButton radioButton2 = new RadioButton("stackoverflow is ok :|");
radioButton2.setToggleGroup(group);

RadioButton radioButton3 = new RadioButton("stackoverflow is useless :(");
radioButton3.setToggleGroup(group);
```

Events for Radio Buttons

Typically, when one of the `RadioButton`s in a `ToggleGroup` is selected the application performs an action. Below is an example which prints the user data of the selected `RadioButton` which has been set with `setUserData(Object)`.

```
radioButton1.setUserData("awesome")
radioButton2.setUserData("ok");
radioButton3.setUserData("useless");

ToggleGroup group = new ToggleGroup();
group.selectedToggleProperty().addListener((obserableValue, old_toggle, new_toggle) -> {
    if (group.getSelectedToggle() != null) {
        System.out.println("You think that stackoverflow is " +
group.getSelectedToggle().getUserData().toString());
    }
});
```

Requesting focus for Radio Buttons

Let's say the second `RadioButton` out of three is pre-selected with `setSelected(Boolean)`, the focus is still at the first `RadioButton` by default. To change this use the `requestFocus()` method.

```
radioButton2.setSelected(true);
radioButton2.requestFocus();
```

Read Radio Button online: https://riptutorial.com/javafx/topic/5906/radio-button

Chapter 16: Scene Builder

Introduction

JavaFX Scene Builder is a visual layout tool that lets users quickly design JavaFX application user interfaces, without coding. It is used to generate FXML files.

Remarks

JavaFX Scene Builder is a visual layout tool that lets users quickly design JavaFX application user interfaces, without coding. Users can drag and drop UI components to a work area, modify their properties, apply style sheets, and the FXML code for the layout that they are creating is automatically generated in the background. The result is an FXML file that can then be combined with a Java project by binding the UI to the application's logic.

From a Model View Controller (MVC) perspective:

- The FXML file, containing the description of the user interface, is the view.
- The controller is a Java class, optionally implementing the Initializable class, which is declared as the controller for the FXML file.
- The model consists of domain objects, defined on the Java side, that can be connected to the view through the controller.

Scene Builder Installation

1. Download Scene Builder most recent version from Gluon's website, selecting the installer for your platform or the executable jar.

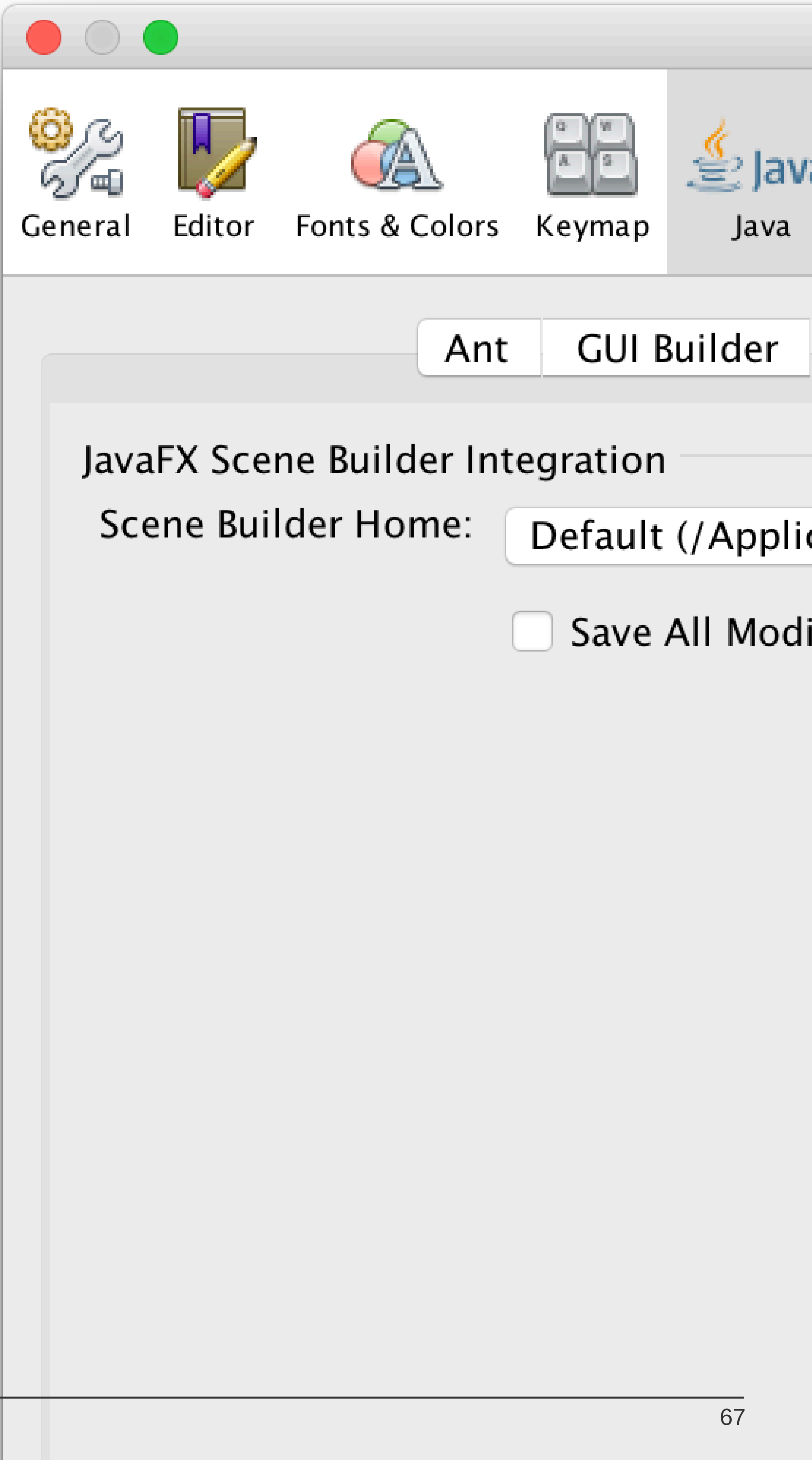
2. With the installer downloaded, double click to install Scene Builder on your system. An updated JRE is included.

3. Double click on the Scene Builder icon to run it as standalone application.

4. IDE Integration

   While Scene Builder is a standalone application, it produces FXML files that are integrated with a Java SE project. When creating this project on an IDE, it is convenient to include a link to the Scene Builder path, so FXML files can be edited.

   - NetBeans: On Windows go to NetBeans->Tools->Options->Java->JavaFX. On Mac OS X go to NetBeans->Preferences->Java->JavaFX. Provide the path for the Scene Builder Home.

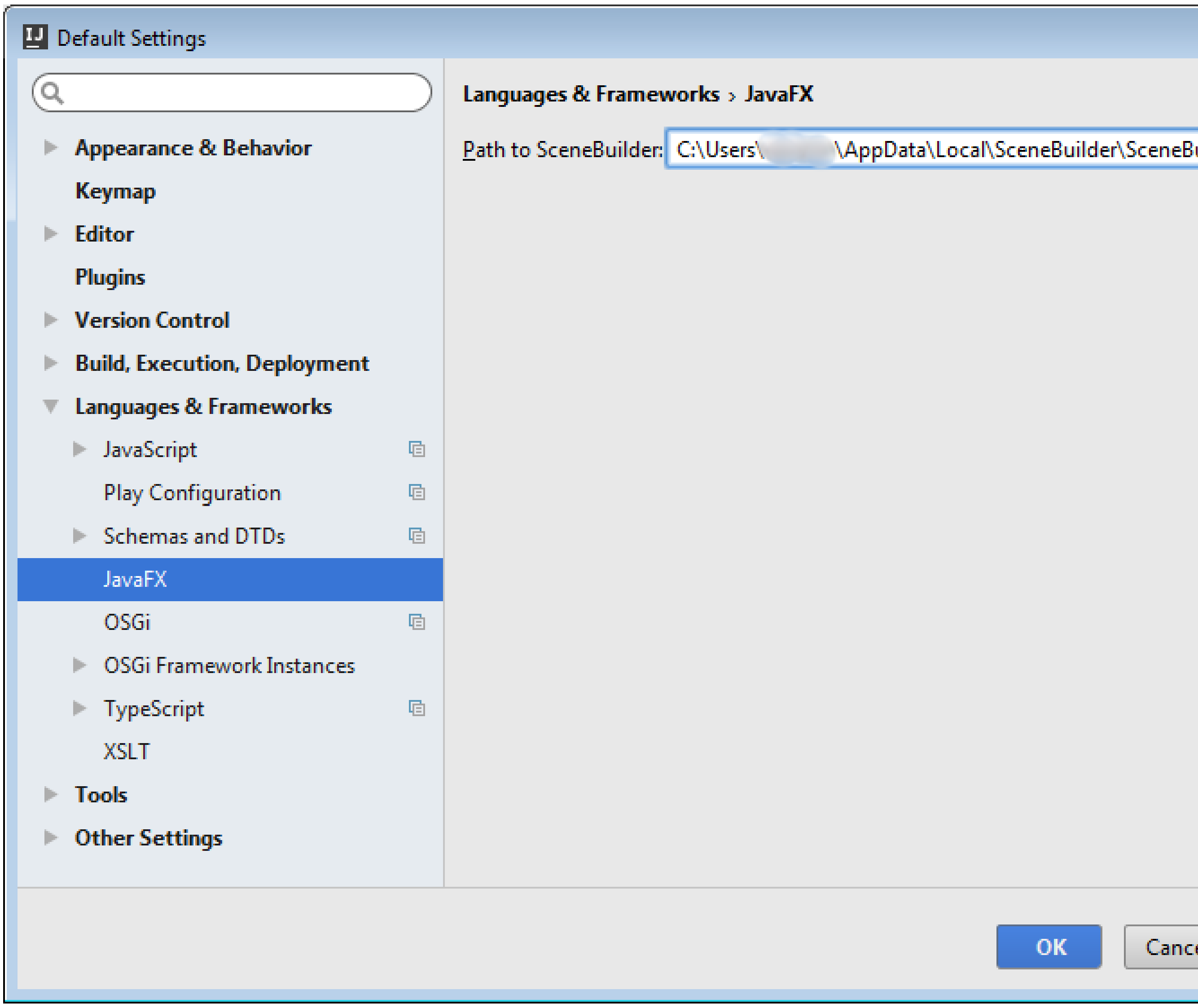


JavaFX Scene Builder Integration

Scene Builder Home: Default (/Applic

Save All Modi

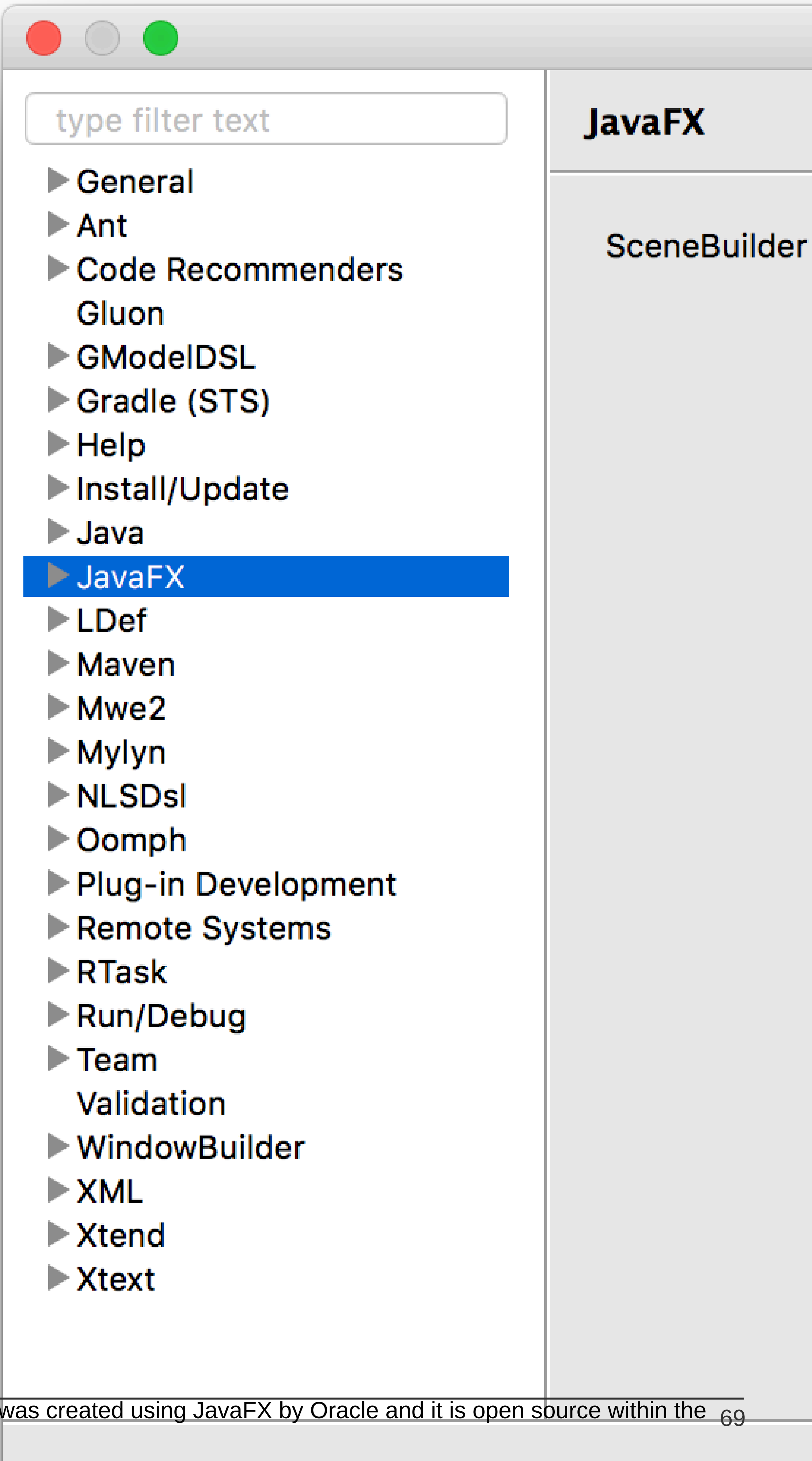
- IntelliJ: On Windows go to IntelliJ->Settings->Languages & Frameworks->JavaFX. On Mac OS X go to IntelliJ->Preferences->Languages & Frameworks->JavaFX. Provide the path for the Scene Builder Home.



Languages & Frameworks › JavaFX

Path to SceneBuilder: C:\Users\ \AppData\Local\SceneBuilder\SceneBu

- Eclipse: On Windows go to Eclipse->Window->Preferences->JavaFX. On Mac OS X go to Eclipse->Preferences->JavaFX. Provide the path for the Scene Builder Home.



A little bit of history

The Scene Builder project was created using JavaFX by Oracle and it is open source within the Oracle provided binaries, up until Scene Builder v 2.0, including only JavaFX features before the release of Java SE 8u40, so new features like the `Spinner` controls are not included.

Gluon took over the binary releases distribution, and an up-to-date Scene Builder 8+ can be downloaded for every platform from here.

It includes the latest changes in JavaFX, and also recent improvements and bug fixes.

The open source project can be found here where issues, feature requests and pull requests can be created.

The Oracle legacy binaries still can be downloaded from here.

Tutorials

Scene Builder tutorials can be found here:

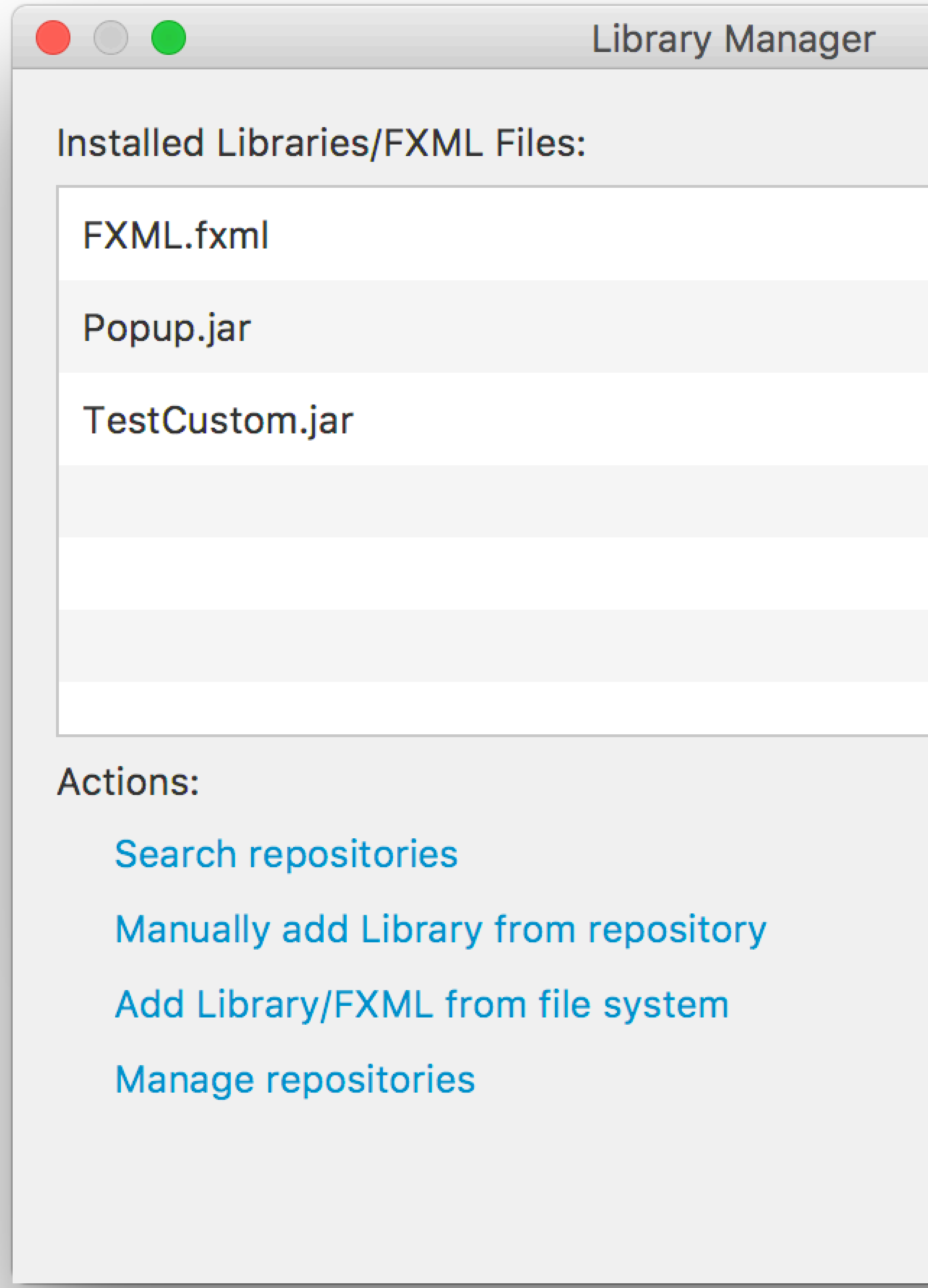
- Oracle Scene Builder 2.0 tutorial

FXML tutorials can be found here.

- Oracle FXML tutorial

Custom controls

Gluon has fully documented the new feature that allows importing third party jars with custom controls, using the Library Manager (available since Scene Builder 8.2.0).


Library Manager

Installed Libraries/FXML Files:

FXML.fxml

Popup.jar

TestCustom.jar

Actions:

Search repositories

Manually add Library from repository

Add Library/FXML from file system

Manage repositories

SO Questions

Tag related: SceneBuilder

from the jar/classpath, as specified by `FXMLLoader.load(getClass().getResource("BasicFXML.fxml"))`.

When loading `basicFXML.fxml`, the loader will find the name of the controller class, as specified by `fx:controller="org.stackoverflow.BasicFXMLController"` in the FXML.

Then the loader will create an instance of that class, in which it will try to inject all the objects that have an `fx:id` in the FXML and are marked with the `@FXML` annotation in the controller class.

In this sample, the FXMLLoader will create the label based on `<Label ... fx:id="label"/>`, and it will inject the label instance into the `@FXML private Label label;`.

Finally, when the whole FXML has been loaded, the FXMLLoader will call the controller's `initialize` method, and the code that registers an action handler with the button will be executed.

Editing

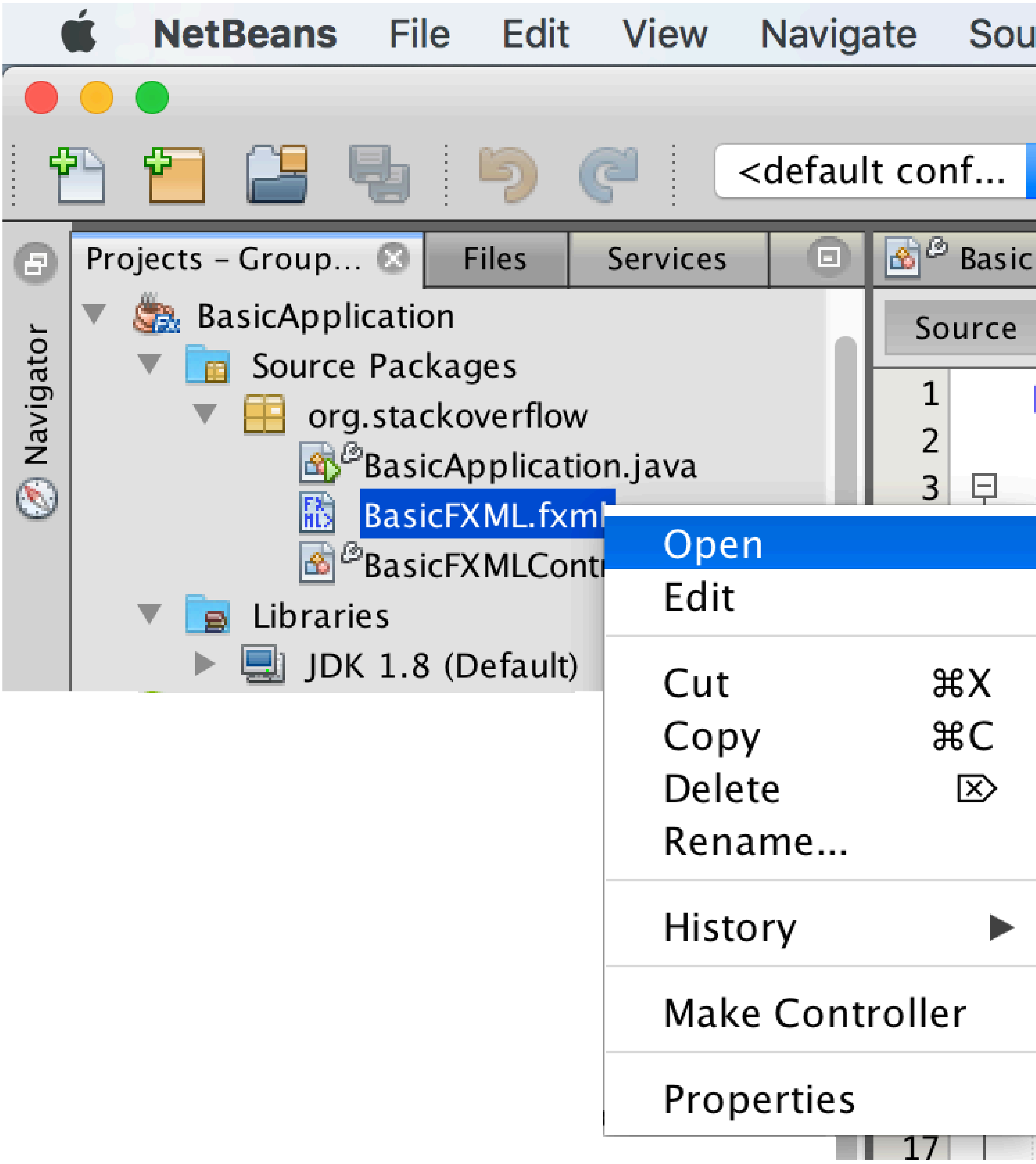
While the FXML file can be edited within the IDE, it is not recommended, as the IDE provides just basic syntax checking and autocompletion, but not visual guidance.

The best approach is opening the FXML file with Scene Builder, where all the changes will be saved to the file.

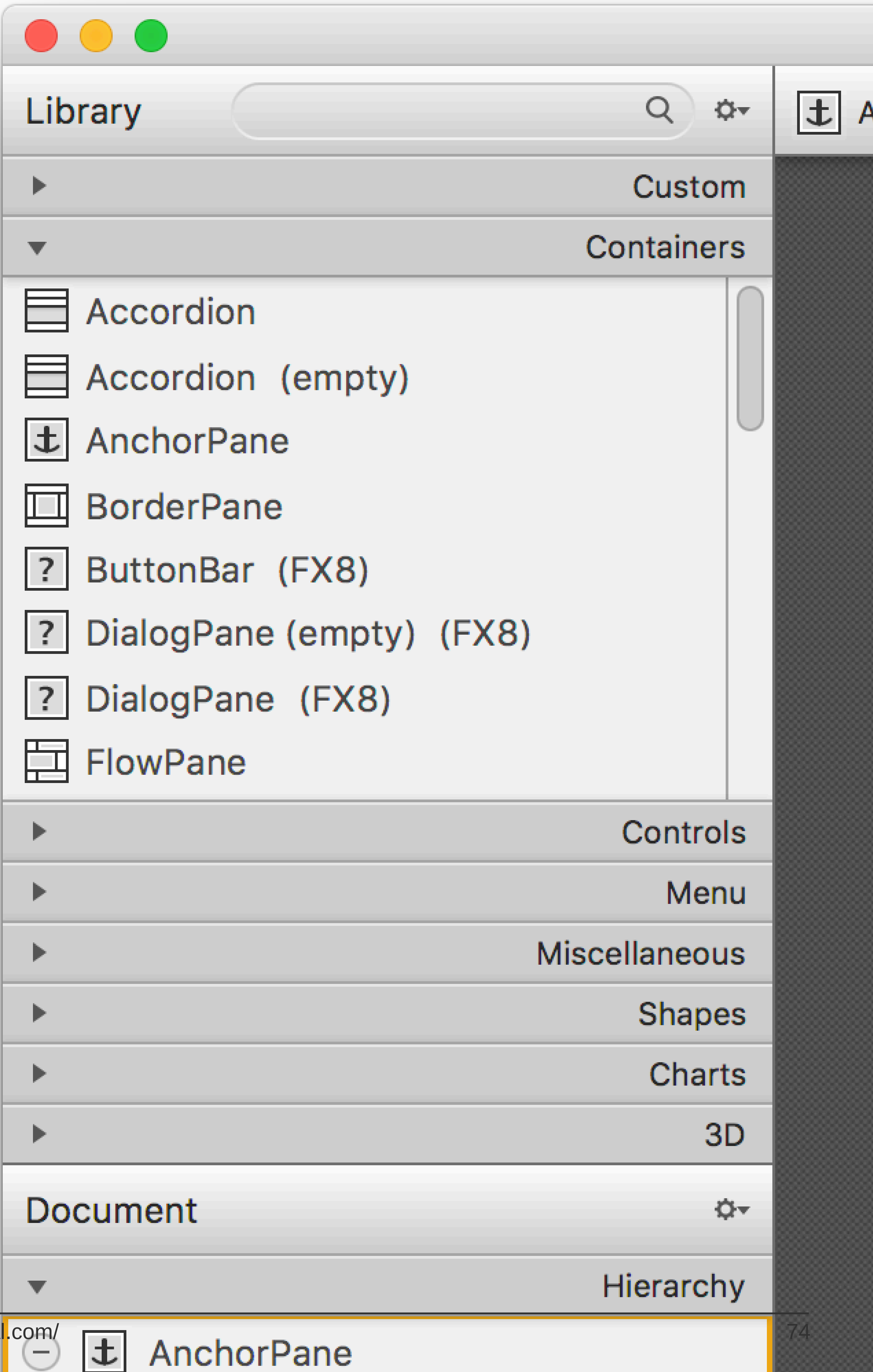
Scene Builder can be launched to open the file:

Or the file can be opened with Scene Builder directly from the IDE:

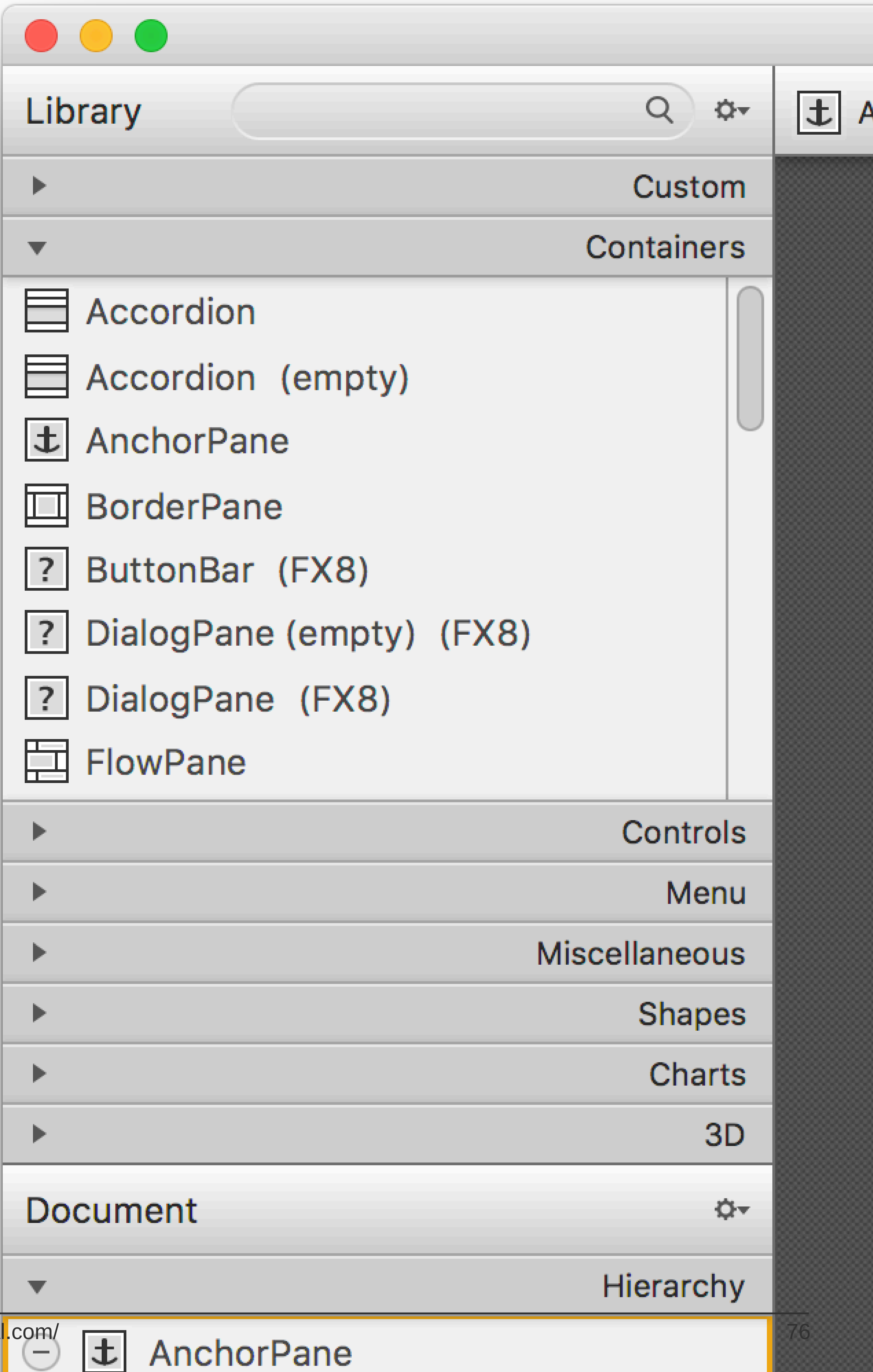
- From NetBeans, on the project tab, double click on the file or right click and select `Open`.
- From IntelliJ, on the project tab, right click on the file and select `Open In Scene Builder`.
- From Eclipse, on the project tab, right click on the file and select `Open with Scene Builder`.



If Scene Builder is properly installed and its path added to the IDE (see Remarks below), it will open the file:


Library
Custom
Containers
Accordion
Accordion (empty)
AnchorPane
BorderPane
ButtonBar (FX8)
DialogPane (empty) (FX8)
DialogPane (FX8)
FlowPane
Controls
Menu
Miscellaneous
Shapes
Charts
3D
Document
Hierarchy
AnchorPane

. It can be set in the `Code` pane:


Library
Custom
Containers
Accordion
Accordion (empty)
AnchorPane
BorderPane
ButtonBar (FX8)
DialogPane (empty) (FX8)
DialogPane (FX8)
FlowPane
Controls
Menu
Miscellaneous
Shapes
Charts
3D
Document
Hierarchy
AnchorPane

Chapter 17: ScrollPane

Introduction

The ScrollPane is a control that offers a dynamic view of its content. This view is controlled in various ways; (increment-decrement button / mouse wheel) to have an integral view of the content.

Examples

A) Fixed content's size :

The size of the content will be the same as that of its ScrollPane container.

```
import javafx.scene.control.ScrollPane;   //Import the ScrollPane
import javafx.scene.control.ScrollPane.ScrollBarPolicy; //Import the ScrollBarPolicy
import javafx.scene.layout.Pane;

ScrollPane scrollpane;
Pane content = new Pane();  //We will use this Pane as a content

scrollpane = new ScrollPane(content);  //Initialize and add content as a parameter
scrollpane.setPrefSize(300, 300);   //Initialize the size of the ScrollPane

scrollpane.setFitToWidth(true);  //Adapt the content to the width of ScrollPane
scrollpane.setFitToHeight(true); //Adapt the content to the height of ScrollPane


scrollpane.setHbarPolicy(ScrollBarPolicy.ALWAYS);  //Control the visibility of the Horizontal
ScrollBar
scrollpane.setVbarPolicy(ScrollBarPolicy.NEVER);  //Control the visibility of the Vertical
ScrollBar
//There are three types of visibility (ALWAYS/AS_NEEDED/NEVER)
```

B) Dynamic content's size :

The size of the content will change depending on the added elements that exceed the content limits in both axes (horizontal and vertical) that can be seen by moving through the view.

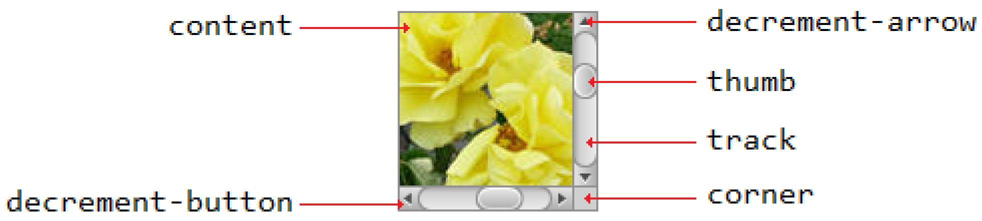
```
import javafx.scene.control.ScrollPane;   //Import the ScrollPane
import javafx.scene.control.ScrollPane.ScrollBarPolicy; //Import the ScrollBarPolicy
import javafx.scene.layout.Pane;

ScrollPane scrollpane;
Pane content = new Pane();  //We will use this Pane as a content

scrollpane = new ScrollPane();
scrollpane.setPrefSize(300, 300);   //Initialize the size of the ScrollPane
content.setMinSize(300,300); //Here a minimum size is set so that the container can be
extended.
scrollpane.setContent(content); // we add the content to the ScrollPane
```

Note : Here we don't need both methods (setFitToWidth/setFitToHeight).

Styling the ScrollPane :

The appearance of the ScrollPane can be easily changed, by having some notions of "*CSS*" and respecting some control "*properties*" and of course having some "*imagination*".

A) The elements that make up ScrollPane :



B) CSS properties :

```
.scroll-bar:vertical .track{}

.scroll-bar:horizontal .track{}

.scroll-bar:horizontal .thumb{}

.scroll-bar:vertical .thumb{}


.scroll-bar:vertical *.increment-button,
.scroll-bar:vertical *.decrement-button{}

.scroll-bar:vertical *.increment-arrow .content,
.scroll-bar:vertical *.decrement-arrow .content{}

.scroll-bar:vertical *.increment-arrow,
.scroll-bar:vertical *.decrement-arrow{}


.scroll-bar:horizontal *.increment-button,
.scroll-bar:horizontal *.decrement-button{}

.scroll-bar:horizontal *.increment-arrow .content,
.scroll-bar:horizontal *.decrement-arrow .content{}

.scroll-bar:horizontal *.increment-arrow,
.scroll-bar:horizontal *.decrement-arrow{}

.scroll-pane .corner{}

.scroll-pane{}
```

Read ScrollPane online: https://riptutorial.com/javafx/topic/8259/scrollpane

Chapter 18: TableView

Examples

Sample TableView with 2 columns

Table Item

The following class contains 2 properties a name (`String`) and the size (`double`). Both properties are wrapped in JavaFX properties to allow the `TableView` to observe changes.

```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;

public class Person {

    public Person(String name, double size) {
        this.size = new SimpleDoubleProperty(this, "size", size);
        this.name = new SimpleStringProperty(this, "name", name);
    }

    private final StringProperty name;
    private final DoubleProperty size;

    public final String getName() {
        return this.name.get();
    }

    public final void setName(String value) {
        this.name.set(value);
    }

    public final StringProperty nameProperty() {
        return this.name;
    }

    public final double getSize() {
        return this.size.get();
    }

    public final void setSize(double value) {
        this.size.set(value);
    }

    public final DoubleProperty sizeProperty() {
        return this.size;
    }

}
```

Sample Application

This application shows a `TableView` with 2 columns; one for the name and one for the size of a

`Person`. Selecting one of the `Person`s adds the data to `TextField`s below the `TableView` and allow the user to edit the data. Note once the edit is commited, the `TableView` is automatically updated.

To every for every `TableColumn` added to the `TableView` a `cellValueFactory` is assigned. This factory is responsible for converting table items (`Person`s) to `ObservableValue`s that contain the value that should be displayed in the table cell and that allows the `TableView` to listen to any changes for this value.

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.TextFormatter;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.Callback;
import javafx.util.StringConverter;

public class TableSample extends Application {

    @Override
    public void start(Stage primaryStage) {
        // data for the tableview. modifying this list automatically updates the tableview
        ObservableList<Person> data = FXCollections.observableArrayList(
                new Person("John Doe", 1.75),
                new Person("Mary Miller", 1.70),
                new Person("Frank Smith", 1.80),
                new Person("Charlotte Hoffman", 1.80)
        );

        TableView<Person> tableView = new TableView<>(data);

        // table column for the name of the person
        TableColumn<Person, String> nameColumn = new TableColumn<>("Name");
        nameColumn.setCellValueFactory(new Callback<TableColumn.CellDataFeatures<Person,
String>, ObservableValue<String>>() {

            @Override
            public ObservableValue<String> call(TableColumn.CellDataFeatures<Person, String>
param) {
                return param.getValue().nameProperty();
            }
        });

        // column for the size of the person
        TableColumn<Person, Number> sizeColumn = new TableColumn<>("Size");
        sizeColumn.setCellValueFactory(new Callback<TableColumn.CellDataFeatures<Person,
Number>, ObservableValue<Number>>() {
```

```
            @Override
            public ObservableValue<Number> call(TableColumn.CellDataFeatures<Person, Number>
param) {
                return param.getValue().sizeProperty();
            }
        });

        // add columns to tableview
        tableView.getColumns().addAll(nameColumn, sizeColumn);

        TextField name = new TextField();

        TextField size = new TextField();

        // convert input from textfield to double
        TextFormatter<Double> sizeFormatter = new TextFormatter<Double>(new
StringConverter<Double>() {

            @Override
            public String toString(Double object) {
                return object == null ? "" : object.toString();
            }

            @Override
            public Double fromString(String string) {
                if (string == null || string.isEmpty()) {
                    return null;
                } else {
                    try {
                        double val = Double.parseDouble(string);
                        return val < 0 ? null : val;
                    } catch (NumberFormatException ex) {
                        return null;
                    }
                }
            }

        });
        size.setTextFormatter(sizeFormatter);

        Button commit = new Button("Change Item");
        commit.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                Person p = tableView.getSelectionModel().getSelectedItem();
                p.setName(name.getText());
                Double value = sizeFormatter.getValue();
                p.setSize(value == null ? -1d : value);
            }

        });

        // listen for changes in the selection to update the data in the textfields
        tableView.getSelectionModel().selectedItemProperty().addListener(new
ChangeListener<Person>() {

            @Override
            public void changed(ObservableValue<? extends Person> observable, Person oldValue,
Person newValue) {
                commit.setDisable(newValue == null);
                if (newValue != null) {
                    sizeFormatter.setValue(newValue.getSize());
                    name.setText(newValue.getName());
                }
            }

        });

        HBox editors = new HBox(5, new Label("Name:"), name, new Label("Size: "), size,
commit);

        VBox root = new VBox(10, tableView, editors);

        Scene scene = new Scene(root);

        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

}
```

PropertyValueFactory

`PropertyValueFactory` can be used as `cellValueFactory` in a `TableColumn`. It uses reflection to access methods that match a certain pattern to retrieve the data from a `TableView` item:

Example

```
TableColumn<Person, String> nameColumn = ...
PropertyValueFactory<Person, String> valueFactory = new PropertyValueFactory<>("name");
nameColumn.setCellValueFactory(valueFactory);
```

The name of the method that is used to get the data depends on the constructor paramerter for `PropertyValueFactory`.

- **Property method:** This kind of method is expected to return a `ObservableValue` containing the data. Changes can be observed. They need to match the pattern `<constructor parameter>Property` and take no parameters.
- **Getter method:** This kind of method expects to return the value directly (`String` in the above example). The method name needs to match the pattern `get<Constructor parameter>`. Note that here `<Constructor parameter>` begins with a *uppercase letter*. This method shouldn't take parameters.

Sample names of methods

constructor parameter (without quotes)	name of property method	name of getter method
foo	fooProperty	getFoo

constructor parameter (without quotes)	name of property method	name of getter method
fooBar	fooBarProperty	getFooBar
XYZ	XYZProperty	getXYZ
listIndex	listIndexProperty	getListIndex
aValue	aValueProperty	getAValue

Customizing TableCell look depending on item

Sometimes a column should show different content than just the `toString` value of the cell item. In this case the `TableCell`s created by the `cellFactory` of the `TableColumn` is customized to change the layout based on the item.

Important Note: `TableView` only creates the `TableCell`s that are shown in the UI. The items inside the cells can change and even become empty. The programmer needs to take care to undo any changes to the `TableCell` that were done when a item was added when it's removed. Otherwise content may still be displayed in a cell where "it doesn't belong".

In the below example setting an item results in the text being set as well as the image displayed in the `ImageView`:

```
image.setImage(item.getEmoji());
setText(item.getValue());
```

If the item becomes `null` or the cell becomes empty, those changes are undone by setting the values back to `null`:

```
setText(null);
image.setImage(null);
```

The following example shows a emoji in addition to text in a `TableCell`.

The `updateItem` method is called every time the item of a `Cell` is changed. Overriding this method allows to react to changes and adjust the look of the cell. Adding a listener to the `itemProperty()` of a cell would be an alternative, but in many cases `TableCell` is extended.

Item type

```
import javafx.scene.image.Image;

// enum providing image and text for certain feelings
public enum Feeling {
    HAPPY("happy",
"https://upload.wikimedia.org/wikipedia/commons/thumb/8/80/Emojione_1F600.svg/64px-
Emojione_1F600.svg.png"),
    SAD("sad",
"https://upload.wikimedia.org/wikipedia/commons/thumb/4/42/Emojione_1F62D.svg/64px-
Emojione_1F62D.svg.png")
    ;
    private final Image emoji;
    private final String value;

    Feeling(String value, String url) {
        // load image in background
        emoji = new Image(url, true);
        this.value = value;
    }

    public Image getEmoji() {
        return emoji;
    }

    public String getValue() {
        return value;
    }

}
```

Code in Application class

```
import javafx.application.Application;
import javafx.beans.property.ObjectProperty;
import javafx.beans.property.SimpleObjectProperty;
import javafx.collections.FXCollections;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TableCell;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.Callback;

public class EmotionTable extends Application {

    public static class Item {

        private final ObjectProperty<Feeling> feeling;

        public Item(Feeling feeling) {
            this.feeling = new SimpleObjectProperty<>(feeling);
        }

        public final Feeling getFeeling() {
            return this.feeling.get();
        }

        public final void setFeeling(Feeling value) {
            this.feeling.set(value);
        }
        public final ObjectProperty<Feeling> feelingProperty() {
            return this.feeling;
        }

    }

    @Override
    public void start(Stage primaryStage) {
        TableView<Item> table = new TableView<>(FXCollections.observableArrayList(
                new Item(Feeling.HAPPY),
                new Item(Feeling.HAPPY),
                new Item(Feeling.HAPPY),
                new Item(Feeling.SAD),
                null,
                new Item(Feeling.HAPPY),
                new Item(Feeling.HAPPY),
                new Item(Feeling.SAD)
        ));

        EventHandler<ActionEvent> eventHandler = new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                // change table items depending on userdata of source
                Node source = (Node) event.getSource();
                Feeling targetFeeling = (Feeling) source.getUserData();
                for (Item item : table.getItems()) {
                    if (item != null) {
                        item.setFeeling(targetFeeling);
                    }
                }
            }

        };

        TableColumn<Item, Feeling> feelingColumn = new TableColumn<>("Feeling");

        feelingColumn.setCellValueFactory(new PropertyValueFactory<>("feeling"));

        // use custom tablecell to display emoji image
        feelingColumn.setCellFactory(new Callback<TableColumn<Item, Feeling>, TableCell<Item,
Feeling>>() {

            @Override
            public TableCell<Item, Feeling> call(TableColumn<Item, Feeling> param) {
                return new EmojiCell<>();
            }
        });

        table.getColumns().add(feelingColumn);

        Button sunshine = new Button("sunshine");
        Button rain = new Button("rain");

        sunshine.setOnAction(eventHandler);
        rain.setOnAction(eventHandler);

        sunshine.setUserData(Feeling.HAPPY);
        rain.setUserData(Feeling.SAD);
        Scene scene = new Scene(new VBox(10, table, new HBox(10, sunshine, rain)));

        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

}
```

Cell class

```
import javafx.scene.control.TableCell;
import javafx.scene.image.ImageView;

public class EmojiCell<T> extends TableCell<T, Feeling> {

    private final ImageView image;

    public EmojiCell() {
        // add ImageView as graphic to display it in addition
        // to the text in the cell
        image = new ImageView();
        image.setFitWidth(64);
        image.setFitHeight(64);
        image.setPreserveRatio(true);

        setGraphic(image);
        setMinHeight(70);
    }

    @Override
    protected void updateItem(Feeling item, boolean empty) {
        super.updateItem(item, empty);

        if (empty || item == null) {
            // set back to look of empty cell
            setText(null);
            image.setImage(null);
        } else {
            // set image and text for non-empty cell
            image.setImage(item.getEmoji());
            setText(item.getValue());
        }
    }
}
```

Add Button to Tableview

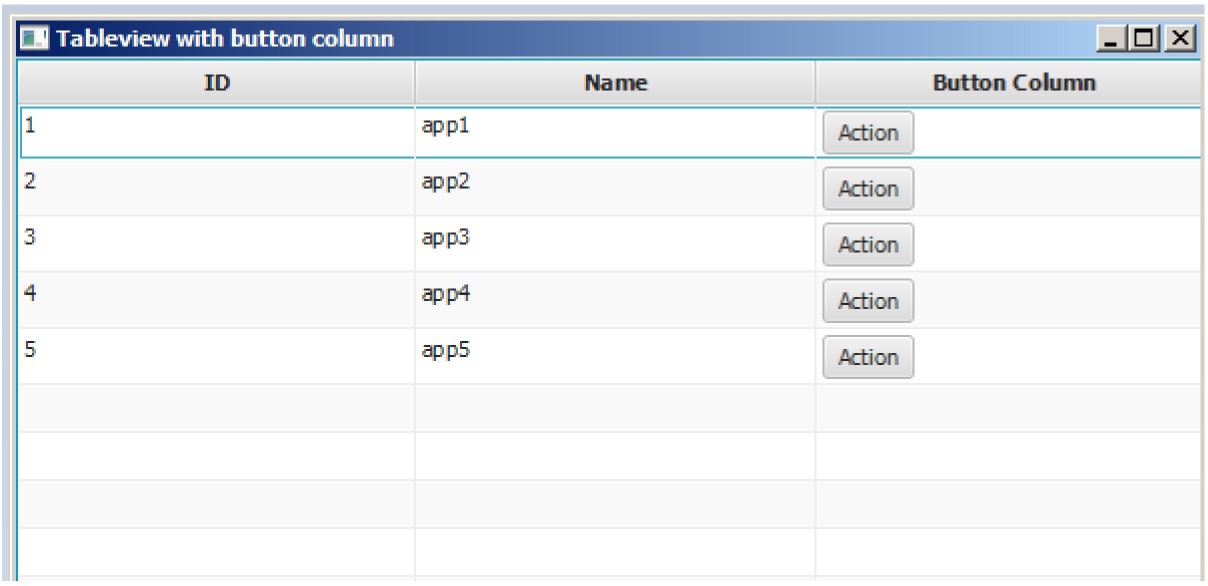
You can add a button or another javafx component to Tableview using column `setCellFactory(Callback value)` method.

Sample Application

In this application we are going to add a button to TableView. When clicked to this column button, data on the same row as button is selected and its information printed.

In the `addButtonToTable()` method, `cellFactory` callback is responsible adding button to related column. We define the callable cellFactory and implement its override `call(...)` method to get `TableCell` with button and then this `cellFactory` set to related column `setCellFactory(..)` method. In our sample this is `colBtn.setCellFactory(cellFactory)`. SSCCE is below:

```
import javafx.application.Application;
import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TableCell;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.stage.Stage;
import javafx.util.Callback;

public class TableViewSample extends Application {

    private final TableView<Data> table = new TableView<>();
    private final ObservableList<Data> tvObservableList = FXCollections.observableArrayList();

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

    @Override
    public void start(Stage stage) {

        stage.setTitle("Tableview with button column");
        stage.setWidth(600);
        stage.setHeight(600);

        setTableappearance();

        fillTableObservableListWithSampleData();
        table.setItems(tvObservableList);

        TableColumn<Data, Integer> colId = new TableColumn<>("ID");
        colId.setCellValueFactory(new PropertyValueFactory<>("id"));

        TableColumn<Data, String> colName = new TableColumn<>("Name");
        colName.setCellValueFactory(new PropertyValueFactory<>("name"));

        table.getColumns().addAll(colId, colName);

        addButtonToTable();

        Scene scene = new Scene(new Group(table));

        stage.setScene(scene);
        stage.show();
    }
    private void setTableappearance() {
        table.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
        table.setPrefWidth(600);
        table.setPrefHeight(600);
    }

    private void fillTableObservableListWithSampleData() {

        tvObservableList.addAll(new Data(1, "app1"),
                                new Data(2, "app2"),
                                new Data(3, "app3"),
                                new Data(4, "app4"),
                                new Data(5, "app5"));
    }

    private void addButtonToTable() {
        TableColumn<Data, Void> colBtn = new TableColumn("Button Column");

        Callback<TableColumn<Data, Void>, TableCell<Data, Void>> cellFactory = new
Callback<TableColumn<Data, Void>, TableCell<Data, Void>>() {
            @Override
            public TableCell<Data, Void> call(final TableColumn<Data, Void> param) {
                final TableCell<Data, Void> cell = new TableCell<Data, Void>() {

                    private final Button btn = new Button("Action");

                    {
                        btn.setOnAction((ActionEvent event) -> {
                            Data data = getTableView().getItems().get(getIndex());
                            System.out.println("selectedData: " + data);
                        });
                    }

                    @Override
                    public void updateItem(Void item, boolean empty) {
                        super.updateItem(item, empty);
                        if (empty) {
                            setGraphic(null);
                        } else {
                            setGraphic(btn);
                        }
                    }
                };
                return cell;
            }
        };

        colBtn.setCellFactory(cellFactory);

        table.getColumns().add(colBtn);

    }

    public class Data {

        private int id;
        private String name;

        private Data(int id, String name) {
            this.id = id;
            this.name = name;
        }

        public int getId() {
            return id;
        }

        public void setId(int ID) {
            this.id = ID;
        }

        public String getName() {
            return name;
        }

        public void setName(String nme) {
            this.name = nme;
        }

        @Override
        public String toString() {
            return "id: " + id + " - " + "name: " + name;
        }

    }
}
```

Screenshot: 

Tableview with button column

ID	Name	Button Column
1	app1	Action
2	app2	Action
3	app3	Action
4	app4	Action
5	app5	Action

Read TableView online: https://riptutorial.com/javafx/topic/2229/tableview

Chapter 19: Threading

Examples

Updating the UI using Platform.runLater

Long-running operations must not be run on the JavaFX application thread, since this prevents JavaFX from updating the UI, resulting in a frozen UI.

Furthermore any change to a `Node` that is part of a "live" scene graph **must** happen on the JavaFX application thread. `Platform.runLater` can be used to execute those updates on the JavaFX application thread.

The following example demonstrates how to update a `Text` `Node` repeatedly from a different thread:

```
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class CounterApp extends Application {

    private int count = 0;
    private final Text text = new Text(Integer.toString(count));

    private void incrementCount() {
        count++;
        text.setText(Integer.toString(count));
    }

    @Override
    public void start(Stage primaryStage) {
        StackPane root = new StackPane();
        root.getChildren().add(text);

        Scene scene = new Scene(root, 200, 200);

        // longrunning operation runs on different thread
        Thread thread = new Thread(new Runnable() {

            @Override
            public void run() {
                Runnable updater = new Runnable() {

                    @Override
                    public void run() {
                        incrementCount();
                    }
                };

                while (true) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException ex) {
                    }

                    // UI update is run on the Application thread
                    Platform.runLater(updater);
                }
            }

        });
        // don't let thread prevent JVM shutdown
        thread.setDaemon(true);
        thread.start();

        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

}
```

Grouping UI updates

The following code makes the UI unresponsive for a short while after the button click, since too many `Platform.runLater` calls are used. (Try scrolling the `ListView` immediately after the button click.)

```
@Override
public void start(Stage primaryStage) {
    ObservableList<Integer> data = FXCollections.observableArrayList();
    ListView<Integer> listView = new ListView<>(data);

    Button btn = new Button("Say 'Hello World'");
    btn.setOnAction((ActionEvent event) -> {
        new Thread(() -> {
            for (int i = 0; i < 100000; i++) {
                final int index = i;
                Platform.runLater(() -> data.add(index));
            }
        }).start();
    });

    Scene scene = new Scene(new VBox(listView, btn));

    primaryStage.setScene(scene);
    primaryStage.show();
}
```

To prevent this instead of using a large number of updates, the following code uses a `AnimationTimer` to run the update only once per frame:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javafx.animation.AnimationTimer;

public class Updater {

    @FunctionalInterface
    public static interface UpdateTask {

        public void update() throws Exception;
    }

    private final List<UpdateTask> updates = new ArrayList<>();

    private final AnimationTimer timer = new AnimationTimer() {

        @Override
        public void handle(long now) {
            synchronized (updates) {
                for (UpdateTask r : updates) {
                    try {
                        r.update();
                    } catch (Exception ex) {
                        Logger.getLogger(Updater.class.getName()).log(Level.SEVERE, null, ex);
                    }
                }
                updates.clear();
                stop();
            }
        }
    };

    public void addTask(UpdateTask... tasks) {
        synchronized (updates) {
            updates.addAll(Arrays.asList(tasks));
            timer.start();
        }
    }

}
```

which allows grouping the updates using the `Updater` class:

```
private final Updater updater = new Updater();

...

        // Platform.runLater(() -> data.add(index));
        updater.addTask(() -> data.add(index));
```

How to use JavaFX Service

Instead of running intensive tasks into `JavaFX Thread` that should be done into a `Service`.So what basically is a Service?

A Service is a class which is creating a new `Thread` every time you are starting it and is passing a Task to it to do some work.The Service can return or not a value.

Below is a typical example of JavaFX Service which is doing some work and return a `Map<String,String>()`:

```
public class WorkerService extends Service<Map<String, String>> {

    /**
     * Constructor
     */
    public WorkerService () {

        // if succeeded
        setOnSucceeded(s -> {
            //code if Service succeeds
        });

        // if failed
        setOnFailed(fail -> {
            //code it Service fails
        });

        //if cancelled
        setOnCancelled(cancelled->{
            //code if Service get's cancelled
        });
    }

    /**
    * This method starts the Service
    */
    public void startTheService(){
        if(!isRunning()){
           //...
           reset();
           start();
        }

    }

    @Override
    protected Task<Map<String, String>> createTask() {
        return new Task<Map<String, String>>() {
            @Override
            protected Void call() throws Exception {

                    //create a Map<String, String>
                    Map<String,String> map  = new HashMap<>();

                   //create other variables here

                   try{
                        //some code here
                        //.....do your manipulation here

                        updateProgress(++currentProgress, totalProgress);
                    }

                } catch (Exception ex) {
                    return null; //something bad happened so you have to do something instead
of returning null
                }
                return map;
            }
        };
    }

}
```

Read Threading online: https://riptutorial.com/javafx/topic/2230/threading

Chapter 20: WebView and WebEngine

Remarks

The `WebView`is the JavaFX Node that is integrated into the JavaFX component tree. It manages a `WebEngine` and displays it's content.

The `WebEngine` is the underlying Browser Engine, which basically does the whole work.

Examples

Loading a page

```
WebView wv = new WebView();
WebEngine we = wv.getEngine();
we.load("https://stackoverflow.com");
```

`WebView` is the UI shell around the `WebEngine`. Nearly all controls for non UI interaction with a page are done through the `WebEngine` class.

Get the page history of a WebView

```
WebHistory history = webView.getEngine().getHistory();
```

The history is basically a list of entries. Each entry represents a visited page and it provides access to relevant page info, such as URL, title, and the date the page was last visited.

The list can be obtained by using the `getEntries()` method. The history and the corresponding list of entries change as `WebEngine` navigates across the web. The list may expand or shrink depending on browser actions. These changes can be listened to by the ObservableList API that the list exposes.

The index of the history entry associated with the currently visited page is represented by the `currentIndexProperty()`. The current index can be used to navigate to any entry in the history by using the `go(int)` method. The `maxSizeProperty()` sets the maximum history size, which is the size of the history list

Below is an example of how to obtain and process the List of Web History Items.

A `ComboBox` (comboBox) is used to store the history items. By using a `ListChangeListener` on the `WebHistory` the `ComboBox` gets updated to the current `WebHistory`. On the `ComboBox` is an `EventHandler` which redirects to the selected page.

```
final WebHistory history = webEngine.getHistory();

comboBox.setItems(history.getEntries());
comboBox.setPrefWidth(60);
comboBox.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent ev) {
        int offset =
                comboBox.getSelectionModel().getSelectedIndex()
                - history.getCurrentIndex();
        history.go(offset);
    }
});

history.currentIndexProperty().addListener(new ChangeListener<Number>() {

    @Override
    public void changed(ObservableValue<? extends Number> observable, Number oldValue, Number
newValue) {
        // update currently selected combobox item
        comboBox.getSelectionModel().select(newValue.intValue());
    }
});

// set converter for value shown in the combobox:
//   display the urls
comboBox.setConverter(new StringConverter<WebHistory.Entry>() {

    @Override
    public String toString(WebHistory.Entry object) {
        return object == null ? null : object.getUrl();
    }

    @Override
    public WebHistory.Entry fromString(String string) {
        throw new UnsupportedOperationException();
    }
});
```

send Javascript alerts from the displayed web page to the Java applications log.

```
private final Logger logger = Logger.getLogger(getClass().getCanonicalName());

WebView webView = new WebView();
webEngine = webView.getEngine();

webEngine.setOnAlert(event -> logger.warning(() -> "JS alert: " + event.getData()));
```

Communication between Java app and Javascript in the web page

When using a WebView to display your own custom webpage and this webpage contains Javascript, it might be necessary to establish a two-way communication between the Java program and the Javascript in the web page.

This example shows how to setup such a communication.

The webpage shall display an input field and a button. On clicking the button, the value from the input field is sent to the Java application, which processes it. After processing a result is sent to the Javascript which in turn displays the result on the web page.

The basic principle is that for communication from Javascript to Java an object is created in Java which is set into the webpage. And for the other direction, an object is created in Javascript and extracted from the webpage.

The following code shows the Java part, I kept it all in one file:

```
package com.sothawo.test;

import javafx.application.Application;
import javafx.concurrent.Worker;
import javafx.scene.Scene;
import javafx.scene.web.WebEngine;
import javafx.scene.web.WebView;
import javafx.stage.Stage;
import netscape.javascript.JSObject;

import java.io.File;
import java.net.URL;

/**
 * @author P.J. Meisch (pj.meisch@sothawo.com).
 */
public class WebViewApplication extends Application {

    /** for communication to the Javascript engine. */
    private JSObject javascriptConnector;

    /** for communication from the Javascript engine. */
    private JavaConnector javaConnector = new JavaConnector();;

    @Override
    public void start(Stage primaryStage) throws Exception {
        URL url = new File("./js-sample.html").toURI().toURL();

        WebView webView = new WebView();
        final WebEngine webEngine = webView.getEngine();

        // set up the listener
        webEngine.getLoadWorker().stateProperty().addListener((observable, oldValue, newValue)
-> {
            if (Worker.State.SUCCEEDED == newValue) {
                // set an interface object named 'javaConnector' in the web engine's page
                JSObject window = (JSObject) webEngine.executeScript("window");
                window.setMember("javaConnector", javaConnector);

                // get the Javascript connector object.
                javascriptConnector = (JSObject) webEngine.executeScript("getJsConnector()");
            }
        });

        Scene scene = new Scene(webView, 300, 150);
        primaryStage.setScene(scene);
        primaryStage.show();

        // now load the page
        webEngine.load(url.toString());
    }
    public class JavaConnector {
        /**
         * called when the JS side wants a String to be converted.
         *
         * @param value
         *         the String to convert
         */
        public void toLowerCase(String value) {
            if (null != value) {
                javascriptConnector.call("showResult", value.toLowerCase());
            }
        }
    }
}
```

When the page has loaded, a `JavaConnector` object (which is defined by the inner class and created as a field) is set into the web page by these calls:

```
JSObject window = (JSObject) webEngine.executeScript("window");
window.setMember("javaConnector", javaConnector);
```

The `javascriptConnector` object is retrieved from the webpage with

```
javascriptConnector = (JSObject) webEngine.executeScript("getJsConnector()");
```

When the `toLowerCase(String)` method from the `JavaConnector` is called, the passed in value is converted and then sent back via the `javascriptConnector` object.

And this is the html and javascript code:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Sample</title>
    </head>
    <body>
        <main>

            <div><input id="input" type="text"></div>
            <button onclick="sendToJava();">to lower case</button>
            <div id="result"></div>

        </main>

        <script type="text/javascript">
            function sendToJava () {
                var s = document.getElementById('input').value;
                javaConnector.toLowerCase(s);
            };

            var jsConnector = {
                showResult: function (result) {
                    document.getElementById('result').innerHTML = result;
                }
            };

            function getJsConnector() {
                return jsConnector;
            };
        </script>
    </body>
</html>
```

The `sendToJava` function calls the method of the `JavaConnector` which was set by the Java code:

```
function sendToJava () {
    var s = document.getElementById('input').value;
    javaConnector.toLowerCase(s);
};
```

and the function called by the Java code to retrieve the `javascriptConnector` just returns the `jsConnector` object:

```
var jsConnector = {
    showResult: function (result) {
        document.getElementById('result').innerHTML = result;
    }
};

function getJsConnector() {
    return jsConnector;
};
```

The argument type of the calls between Java and Javascript are not limited to Strings. More info on the possible types and conversion are found in the JSObject API doc.

Read WebView and WebEngine online: https://riptutorial.com/javafx/topic/5156/webview-and-webengine

Chapter 21: Windows

Examples

Creating a new Window

To show some content in a new window, a `Stage` needs to be created. After creation and initialisation `show` or `showAndWait` needs to be called on the `Stage` object:

```
// create sample content
Rectangle rect = new Rectangle(100, 100, 200, 300);
Pane root = new Pane(rect);
root.setPrefSize(500, 500);

Parent content = root;

// create scene containing the content
Scene scene = new Scene(content);

Stage window = new Stage();
window.setScene(scene);

// make window visible
window.show();
```

Note: This code needs to be executed on the JavaFX application thread.

Creating Custom Dialog

You can create custom dialogs which contains many component and perform many functionality on it. It behaves like second stage on owner stage.
In the following example an application that shows person in the main stage tableview and creates a person in a dialog (AddingPersonDialog) prepared. GUIs created by SceneBuilder, but they can be created by pure java codes.

Sample Application:

AppMain.java

```
package customdialog;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class AppMain extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("AppMain.fxml"));
        Scene scene = new Scene(root, 500, 500);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

}
```

AppMainController.java

```
package customdialog;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.fxml.Initializable;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.stage.Modality;
import javafx.stage.Stage;

public class AppMainController implements Initializable {

    @FXML
    private TableView<Person> tvData;
    @FXML
    private TableColumn colId;
    @FXML
    private TableColumn colName;
    @FXML
    private TableColumn colAge;

    private ObservableList<Person> tvObservableList = FXCollections.observableArrayList();

    @FXML
    void onOpenDialog(ActionEvent event) throws IOException {
        FXMLLoader fxmlLoader = new
FXMLLoader(getClass().getResource("AddPersonDialog.fxml"));
        Parent parent = fxmlLoader.load();
        AddPersonDialogController dialogController =
fxmlLoader.<AddPersonDialogController>getController();
        dialogController.setAppMainObservableList(tvObservableList);

        Scene scene = new Scene(parent, 300, 200);
        Stage stage = new Stage();
        stage.initModality(Modality.APPLICATION_MODAL);
        stage.setScene(scene);
        stage.showAndWait();
    }

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        colId.setCellValueFactory(new PropertyValueFactory<>("id"));
        colName.setCellValueFactory(new PropertyValueFactory<>("name"));
        colAge.setCellValueFactory(new PropertyValueFactory<>("age"));
        tvData.setItems(tvObservableList);
    }

}
```

AppMain.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TableColumn?>
<?import javafx.scene.control.TableView?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.VBox?>

<AnchorPane maxHeight="400.0" minHeight="400.0" minWidth="500.0"
xmlns="http://javafx.com/javafx/8.0.111" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="customdialog.AppMainController">
   <children>
      <VBox alignment="CENTER" layoutX="91.0" layoutY="85.0" spacing="10.0"
AnchorPane.bottomAnchor="30.0" AnchorPane.leftAnchor="30.0" AnchorPane.rightAnchor="30.0"
AnchorPane.topAnchor="30.0">
         <children>
            <Button mnemonicParsing="false" onAction="#onOpenDialog" text="Add Person" />
            <TableView fx:id="tvData" prefHeight="300.0" prefWidth="400.0">
              <columns>
                <TableColumn fx:id="colId" prefWidth="75.0" text="ID" />
                <TableColumn fx:id="colName" prefWidth="75.0" text="Name" />
                  <TableColumn fx:id="colAge" prefWidth="75.0" text="Age" />
              </columns>
               <columnResizePolicy>
                  <TableView fx:constant="CONSTRAINED_RESIZE_POLICY" />
               </columnResizePolicy>
            </TableView>
         </children>
      </VBox>
   </children>
</AnchorPane>
```

AddPersonDialogController.java

```
package customdialog;

import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.Node;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

public class AddPersonDialogController  {

    @FXML
    private TextField tfId;
    @FXML
    private TextField tfName;

    @FXML
    private TextField tfAge;

    private ObservableList<Person> appMainObservableList;

    @FXML
    void btnAddPersonClicked(ActionEvent event) {
        System.out.println("btnAddPersonClicked");
        int id = Integer.valueOf(tfId.getText().trim());
        String name = tfName.getText().trim();
        int iAge = Integer.valueOf(tfAge.getText().trim());

        Person data = new Person(id, name, iAge);
        appMainObservableList.add(data);

        closeStage(event);
    }

    public void setAppMainObservableList(ObservableList<Person> tvObservableList) {
        this.appMainObservableList = tvObservableList;

    }

    private void closeStage(ActionEvent event) {
        Node  source = (Node)  event.getSource();
        Stage stage  = (Stage) source.getScene().getWindow();
        stage.close();
    }

}
```

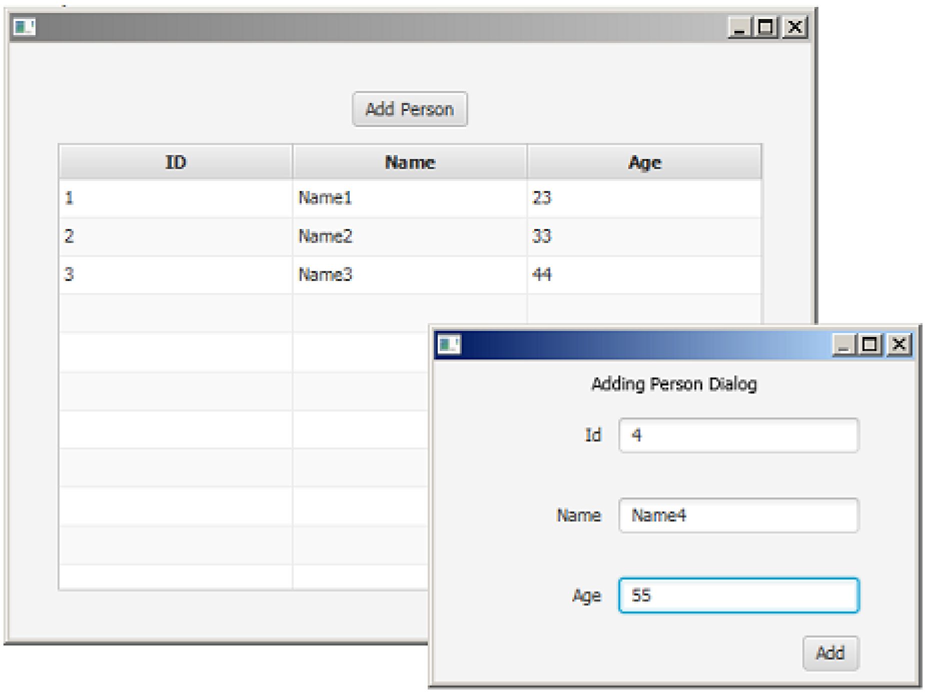
AddPersonDialog.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Text?>

<AnchorPane minHeight="300.0" minWidth="400.0" xmlns="http://javafx.com/javafx/8.0.111"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="customdialog.AddPersonDialogController">
   <children>
      <VBox alignment="CENTER" layoutX="131.0" layoutY="50.0" prefHeight="200.0"
prefWidth="100.0" AnchorPane.bottomAnchor="5.0" AnchorPane.leftAnchor="5.0"
AnchorPane.rightAnchor="5.0" AnchorPane.topAnchor="5.0">
         <children>
            <Text strokeType="OUTSIDE" strokeWidth="0.0" text="Adding Person Dialog" />
            <HBox alignment="CENTER" prefHeight="50.0" prefWidth="200.0" spacing="10.0">
               <children>
                  <Label alignment="CENTER_RIGHT" minWidth="100.0" text="Id" />
                  <TextField fx:id="tfId" HBox.hgrow="ALWAYS" />
               </children>
```

```
                <padding>
                   <Insets right="30.0" />
                </padding>
             </HBox>
             <HBox alignment="CENTER" prefHeight="50.0" prefWidth="200.0" spacing="10.0">
                <children>
                   <Label alignment="CENTER_RIGHT" minWidth="100.0" text="Name" />
                   <TextField fx:id="tfName" HBox.hgrow="ALWAYS" />
                </children>
                <padding>
                   <Insets right="30.0" />
                </padding>
             </HBox>
             <HBox alignment="CENTER" prefHeight="50.0" prefWidth="200.0" spacing="10.0">
                <children>
                   <Label alignment="CENTER_RIGHT" minWidth="100.0" text="Age" />
                   <TextField fx:id="tfAge" HBox.hgrow="ALWAYS" />
                </children>
                <padding>
                   <Insets right="30.0" />
                </padding>
             </HBox>
             <HBox alignment="CENTER_RIGHT">
                <children>
                   <Button mnemonicParsing="false" onAction="#btnAddPersonClicked" text="Add"
/>
                </children>
                <opaqueInsets>
                   <Insets />
                </opaqueInsets>
                <padding>
                   <Insets right="30.0" />
                </padding>
             </HBox>
          </children>
       </VBox>
    </children>
</AnchorPane>
```

Person.java

```
package customdialog;

import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleStringProperty;

public class Person {

    private SimpleIntegerProperty id;
    private SimpleStringProperty name;
    private SimpleIntegerProperty age;

    public Person(int id, String name, int age)  {
        this.id = new SimpleIntegerProperty(id);
        this.name = new SimpleStringProperty(name);
        this.age = new SimpleIntegerProperty(age);
    }

    public int getId() {
        return id.get();
    }

    public void setId(int ID) {
        this.id.set(ID);
    }

    public String getName() {
        return name.get();
    }

    public void setName(String nme) {
        this.name.set(nme);
    }

    public int getAge() {
        return age.get();
    }

    public void setAge(int age) {
        this.age.set(age);
    }

    @Override
    public String toString() {
        return "id: " + id.get() + " - " + "name: " + name.get()+ "age: "+ age.get();
    }

}
```



ID	Name	Age
1	Name1	23
2	Name2	33
3	Name3	44

Id: 4
Name: Name4
Age: 55

Screenshot

Creating Custom Dialog

You can create custom dialogs which contains many component and perform many functionality on it. It behaves like second stage on owner stage.
In the following example an application that shows person in the main stage tableview and creates a person in a dialog (AddingPersonDialog) prepared. GUIs created by SceneBuilder, but they can

be created by pure java codes.

Sample Application:

AppMain.java

```
package customdialog;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class AppMain extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("AppMain.fxml"));
        Scene scene = new Scene(root, 500, 500);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

}
```

AppMainController.java

```
package customdialog;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.fxml.Initializable;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.stage.Modality;
import javafx.stage.Stage;

public class AppMainController implements Initializable {

    @FXML
    private TableView<Person> tvData;
    @FXML
    private TableColumn colId;
    @FXML
    private TableColumn colName;
    @FXML
    private TableColumn colAge;
    private ObservableList<Person> tvObservableList = FXCollections.observableArrayList();

    @FXML
    void onOpenDialog(ActionEvent event) throws IOException {
        FXMLLoader fxmlLoader = new
FXMLLoader(getClass().getResource("AddPersonDialog.fxml"));
        Parent parent = fxmlLoader.load();
        AddPersonDialogController dialogController =
fxmlLoader.<AddPersonDialogController>getController();
        dialogController.setAppMainObservableList(tvObservableList);

        Scene scene = new Scene(parent, 300, 200);
        Stage stage = new Stage();
        stage.initModality(Modality.APPLICATION_MODAL);
        stage.setScene(scene);
        stage.showAndWait();
    }

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        colId.setCellValueFactory(new PropertyValueFactory<>("id"));
        colName.setCellValueFactory(new PropertyValueFactory<>("name"));
        colAge.setCellValueFactory(new PropertyValueFactory<>("age"));
        tvData.setItems(tvObservableList);
    }

}
```

AppMain.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TableColumn?>
<?import javafx.scene.control.TableView?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.VBox?>

<AnchorPane maxHeight="400.0" minHeight="400.0" minWidth="500.0"
xmlns="http://javafx.com/javafx/8.0.111" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="customdialog.AppMainController">
   <children>
      <VBox alignment="CENTER" layoutX="91.0" layoutY="85.0" spacing="10.0"
AnchorPane.bottomAnchor="30.0" AnchorPane.leftAnchor="30.0" AnchorPane.rightAnchor="30.0"
AnchorPane.topAnchor="30.0">
         <children>
            <Button mnemonicParsing="false" onAction="#onOpenDialog" text="Add Person" />
            <TableView fx:id="tvData" prefHeight="300.0" prefWidth="400.0">
              <columns>
                <TableColumn fx:id="colId" prefWidth="75.0" text="ID" />
                <TableColumn fx:id="colName" prefWidth="75.0" text="Name" />
                  <TableColumn fx:id="colAge" prefWidth="75.0" text="Age" />
              </columns>
               <columnResizePolicy>
                  <TableView fx:constant="CONSTRAINED_RESIZE_POLICY" />
               </columnResizePolicy>
            </TableView>
         </children>
      </VBox>
   </children>
```

```
</AnchorPane>
```

AddPersonDialogController.java

```
package customdialog;

import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.Node;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

public class AddPersonDialogController  {

    @FXML
    private TextField tfId;

    @FXML
    private TextField tfName;

    @FXML
    private TextField tfAge;

    private ObservableList<Person> appMainObservableList;

    @FXML
    void btnAddPersonClicked(ActionEvent event) {
        System.out.println("btnAddPersonClicked");
        int id = Integer.valueOf(tfId.getText().trim());
        String name = tfName.getText().trim();
        int iAge = Integer.valueOf(tfAge.getText().trim());

        Person data = new Person(id, name, iAge);
        appMainObservableList.add(data);

        closeStage(event);
    }

    public void setAppMainObservableList(ObservableList<Person> tvObservableList) {
        this.appMainObservableList = tvObservableList;

    }

    private void closeStage(ActionEvent event) {
        Node  source = (Node)  event.getSource();
        Stage stage  = (Stage) source.getScene().getWindow();
        stage.close();
    }

}
```

AddPersonDialog.fxml

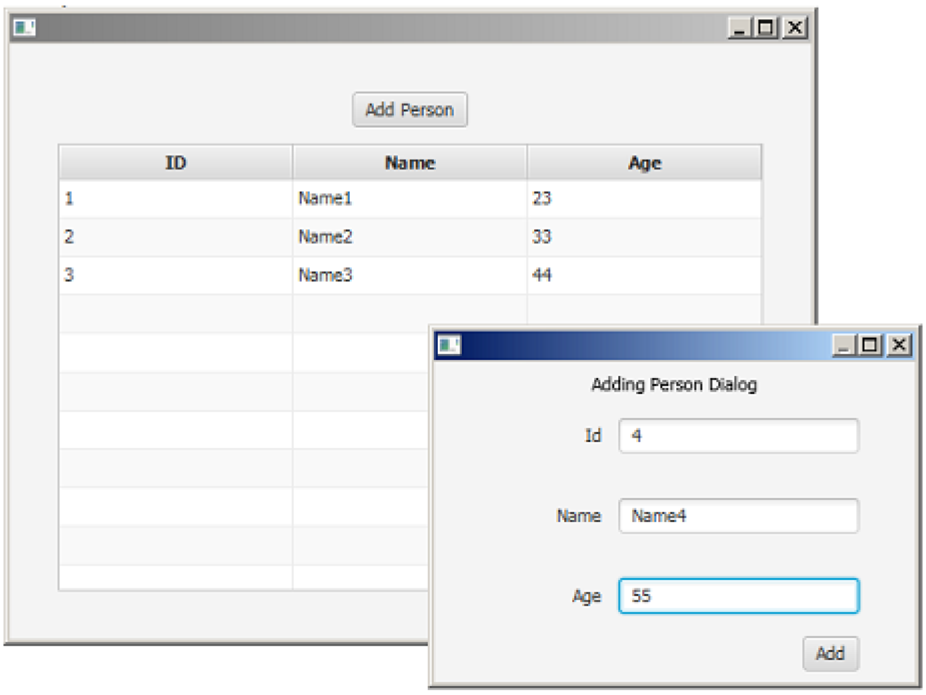
```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
```

```
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.text.Text?>

<AnchorPane minHeight="300.0" minWidth="400.0" xmlns="http://javafx.com/javafx/8.0.111"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="customdialog.AddPersonDialogController">
   <children>
      <VBox alignment="CENTER" layoutX="131.0" layoutY="50.0" prefHeight="200.0"
prefWidth="100.0" AnchorPane.bottomAnchor="5.0" AnchorPane.leftAnchor="5.0"
AnchorPane.rightAnchor="5.0" AnchorPane.topAnchor="5.0">
         <children>
            <Text strokeType="OUTSIDE" strokeWidth="0.0" text="Adding Person Dialog" />
            <HBox alignment="CENTER" prefHeight="50.0" prefWidth="200.0" spacing="10.0">
               <children>
                  <Label alignment="CENTER_RIGHT" minWidth="100.0" text="Id" />
                  <TextField fx:id="tfId" HBox.hgrow="ALWAYS" />
               </children>
               <padding>
                  <Insets right="30.0" />
               </padding>
            </HBox>
            <HBox alignment="CENTER" prefHeight="50.0" prefWidth="200.0" spacing="10.0">
               <children>
                  <Label alignment="CENTER_RIGHT" minWidth="100.0" text="Name" />
                  <TextField fx:id="tfName" HBox.hgrow="ALWAYS" />
               </children>
               <padding>
                  <Insets right="30.0" />
               </padding>
            </HBox>
            <HBox alignment="CENTER" prefHeight="50.0" prefWidth="200.0" spacing="10.0">
               <children>
                  <Label alignment="CENTER_RIGHT" minWidth="100.0" text="Age" />
                  <TextField fx:id="tfAge" HBox.hgrow="ALWAYS" />
               </children>
               <padding>
                  <Insets right="30.0" />
               </padding>
            </HBox>
            <HBox alignment="CENTER_RIGHT">
               <children>
                  <Button mnemonicParsing="false" onAction="#btnAddPersonClicked" text="Add"
/>
               </children>
               <opaqueInsets>
                  <Insets />
               </opaqueInsets>
               <padding>
                  <Insets right="30.0" />
               </padding>
            </HBox>
         </children>
      </VBox>
   </children>
</AnchorPane>
```

Person.java

```
package customdialog;

import javafx.beans.property.SimpleIntegerProperty;
import javafx.beans.property.SimpleStringProperty;

public class Person {

    private SimpleIntegerProperty id;
    private SimpleStringProperty name;
    private SimpleIntegerProperty age;

    public Person(int id, String name, int age)  {
        this.id = new SimpleIntegerProperty(id);
        this.name = new SimpleStringProperty(name);
        this.age = new SimpleIntegerProperty(age);
    }

    public int getId() {
        return id.get();
    }

    public void setId(int ID) {
        this.id.set(ID);
    }

    public String getName() {
        return name.get();
    }

    public void setName(String nme) {
        this.name.set(nme);
    }

    public int getAge() {
        return age.get();
    }

    public void setAge(int age) {
        this.age.set(age);
    }

    @Override
    public String toString() {
        return "id: " + id.get() + " - " + "name: " + name.get()+ "age: "+ age.get();
    }

}
```

Screenshot



ID	Name	Age
1	Name1	23
2	Name2	33
3	Name3	44

Adding Person Dialog

Id	4
Name	Name4
Age	55

Read Windows online: https://riptutorial.com/javafx/topic/1496/windows

Credits

S. No	Chapters	Contributors
1	Getting started with javafx	Community, CraftedCart, D3181, DVarga, fabian, Ganesh, Hendrik Ebbers, Petter Friberg
2	Animation	fabian, J Atkin
3	Button	Dth, DVarga, J Atkin, Maverick283, Nico T, Squidward
4	Canvas	Dth
5	Chart	Dth, James_D, Jinu P C
6	CSS	fabian
7	Dialogs	fabian, GltknBtn, Modus Tollens
8	FXML and Controllers	D3181, fabian, James_D
9	Internationalization in JavaFX	ItachiUchiha, Joffrey, Nico T, P.J.Meisch
10	JavaFX bindings	Alexiy
11	Layouts	DVarga, fabian, Filip Smola, Jinu P C, Sohan Chowdhury, trashgod
12	Pagination	fabian, J Atkin
13	Printing	J Atkin, Squidward
14	Properties & Observable	fabian
15	Radio Button	Nico T
16	Scene Builder	Ashlyn Campbell, José Pereda
17	ScrollPane	Bo Halim
18	TableView	Bo Halim, fabian, GltknBtn
19	Threading	Brendan, fabian, GOXR3PLUS, James_D, Koko Essam, sazzy4o

20	WebView and WebEngine	fabian, J Atkin, James_D, P.J.Meisch, Squidward
21	Windows	fabian, GltknBtn