HOL 1634

"Customize Your JavaFX Controls"

Gerrit Grunwald (Senior Software Engineer, Canoo Engineering AG)

Todd Costella (Technical Architect, Entero Corporation)

Content

1. Overview

When doing frontend development you are always faced with components, no matter what kind of technology you use. Each technology has it's own strategies on how to build those controls. In Java Swing it mainly was extending JComponent and in JavaFX the most "known" way is creating a component by extending Control, Skin and add a CSS file but in JavaFX you have more than one possibility to create your own custom controls.

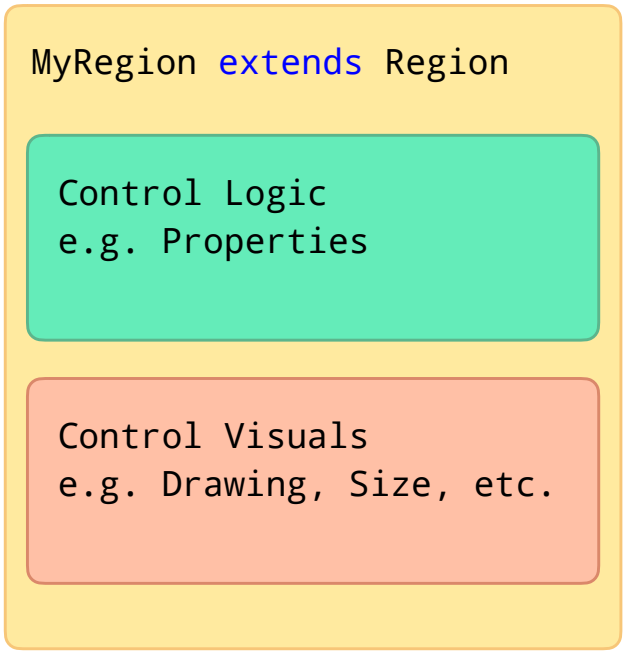
In this Hands on Lab (HOL) session we will show you how to create a custom control by

- Combining existing controls
- Extending existing controls
- Re-Style existing controls
- Extending Region
- Using a Control and Skin

In principle a control is like a tiny version of a whole application, it contains some logic like the backend part and some visualization which is the frontend part. To the user of the control this implementation details are hidden and one only can see/use the public methods like getters and setters.
In JavaFX we have the ability to create a more compact control which contains the logic and the visualization. This is control extends the Region class and is more an all in one approach. The other possibility is to create a control that consists of two classes, a class that extends Control and on that extends Skin.
The drawing on the next page will give you a short overview.

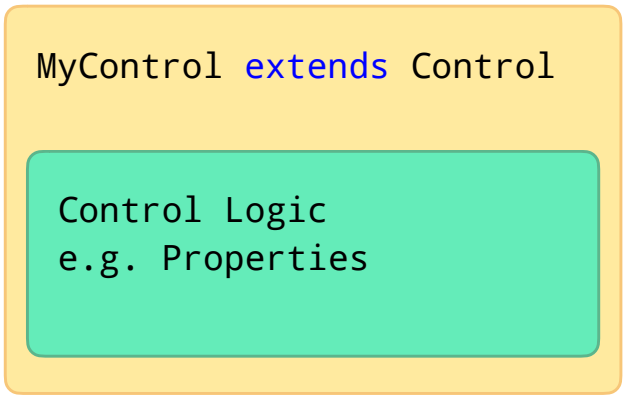
Region based approach

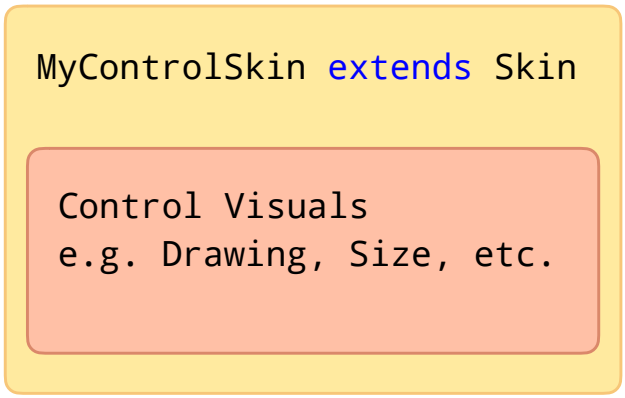
- All in one approach
- CSS support for styling content
- Very flexible
- Contains only the properties that are needed for this control
- Ideal for nearly all controls
- Not really useful for libraries



Control + Skin based approach

- Separation of Logic and Graphics
- CSS support for styling content
- Very flexible
- Can have more than one visualization/Skin
- Contains the properties for all supported Skins
- Ideal for libraries





2. Setup

To complete this HOL session you will need the following setup:

Computer with either Windows, OS X or Linux
Git
Gradle
Java 8
NetBeans, IntelliJ Idea or Eclipse

All the code is also available on github at: http://github.com/HanSolo/

As a software developer you know that coding always also has a personal touch, means when you create code you have your own style and so do I :)
I try to always use the same or similar structure in my controls because it's simply more easy to support in the future. On the next page you will find the structure that I use for my custom controls.

The code shows a template for Region based control because this contains the control logic and the visualization part.

```
@DefaultProperty("children")
public class MyRegion extends Region {
    // ******************** All member variables **********************
    ...

    // ******************** Constructors ******************************
    public MyRegion() {
        // Initializing member variables
        ...
        init();
        initGraphics();
        registerListeners();
    }


    // ******************** Initialization ****************************
    private void init() {...}         // initial sizing

    private void initGraphics() {...} // setup the scene graph

    private void registerListeners() {
        widthProperty().addListener(o -> resize());
        heightProperty().addListener(o -> resize());
        someProperty().addListener(o -> handleControlPropertyChanged("N"));
    }

    // ******************** Methods ***********************************
    @Override public void layoutChildren() {...}

    @Override public ObservableList<Node> getChildren() {...}

    protected void handleControlPropertyChanged(String property) {...}

    public double getValue() {...}
    public void setValue(double value) {...}
    public DoubleProperty valueProperty() {...}


    // ******************** CSS Styleable Properties ******************


    // ******************** Style related *****************************
    @Override public String getUserAgentStylesheet() {...}


    // ******************** Resizing **********************************
    private void resize() {...} // resizing all content
}
```

Let me shortly explain each section to give you an idea what it is used for:

// ********** Constructors **********:
Initialize the member variables and call the init(), initGraphics() and registerListeners() methods

// ********** Initialization **********:
Contains the three methods init(), initGraphics() and registerListeners().
The init() method only contains code to set the initial min-, max- and preferred-size of the control.
The initGraphics() method is very important because here we setup the scene graph once. After that we only make changes to the scene graph but we try to avoid adding or removing nodes to it.
The registerListeners() method is the place where we hook up listeners to all properties that we need to react on. Because sometimes different listeners need to call the same action it makes sense to create a method that only handles the property changes (in my controls this is handleControlPropertyChanged(String property)).

// ********** Methods **********:
Here you will usually find all getters, setters and property methods and methods like the handleControlPropertyChanged() method, the getChildren() method (only when extending Region) and the layoutChildren() method.

// ********** CSS Styleable Properties **********:
If I make use of styleable properties you will find the code that is needed for it in this area

// ********** Style related **********:
This area is mainly used in Control-Skin based controls where you will find methods like createDefaultSkin(), getUserAgentStyleSheet() and getClassCssMetaData() etc.
This methods are needed to define the related CSS stylesheet file and for styleable properties.

// ********** Resizing **********:
This area is used for methods like resize() or resizeText() etc.

Because you might not be familiar with the properties that came with JavaFX let's have a really short look at them on the next page.

From POJO's you know how to use properties in Java, so most of the times it's something like follows:

```
public class MyClass {
    private double value;

    public MyClass() {
        this.value = 0;
    }

    public double getValue() { return value; }

    public void setValue(double value) { this.value = value; }
}
```

When you have used Java Swing before you might also know how properties work there:

```
public class MyClass extends JComponent {
    public  static final String VALUE_PROPERTY = "value";
    private              double value;

    public MyClass() {
        this.value = 0;
    }

    public double getValue() { return value; }

    public void setValue(double value) {
        double oldValue = this.value;
        this.value      = value;
        firePropertyChange(VALUE_PROPERTY, oldValue, this.value);
    }
}

public class App extends JFrame {
    private PropertyChangeListener listener;
    private MyClass                myClass;

    public App() {
        listener = e -> {
            if (SwingControl.VALUE_PROPERTY.equals(e.getPropertyName())) {
                System.out.println("New value: " + e.getNewValue());
            }
        };
        myClass = new MyClass();
        myClass.addPropertyChangeListener(listener);
    }
}
```

With JavaFX new properties came along which in principle are similar to Swing, so the Swing example from last page would look like follows in JavaFX:

```
public class MyClass extends Region {
    private DoubleProperty value;

    public MyClass() {
        value = new SimpleDoubleProperty(0);
    }

    public double getValue() { return this.value.get(); }

    public void setValue(double value) { this.value.set(value); }

    public DoubleProperty valueProperty() { return value; }
}

public class App extends Application {
    private MyClass myClass;

    public App() {
        myClass = new MyClass();
        myClass.valueProperty().addListener((o, ov, nv) -> {
                System.out.println("New value: " + nv);
        });
    }

    public void start(Stage stage) {
        StackPane pane = new StackPane(myClass);
        stage.setScene(new Scene(pane));
        stage.show();
    }
}
```

In principle the DoubleProperty class wraps a primitive double internally and makes it available via it's get() and set() method. In addition it is possible to either add an InvalidationListener or ChangeListener to the DoubleProperty. In addition to the listeners JavaFX introduces binding of properties.
Please find more information about the JavaFX properties at

https://wiki.openjdk.java.net/display/OpenJFX/JavaFX+Property+Architecture

2.1 Combing Controls

The easiest way to create a custom control is to combine existing controls.
We will combine a TextField with a Button where the Button will be used to show a unit and to convert between two units.
The idea is to have a temperature field that can show it's value either in degree celsius or degree fahrenheit.
The button should always show the current unit and when you press it the value of the TextField should be converted to the other unit and the text of the button should also change.

Therefore we will need a JavaFX TextField and a Button that looks like this:

Combined in HBox:

With modified CSS:

What's needed:

Layout container that contains the TextField and the Button
Class that extends HBox
CSS modifications to remove focus of each single control
CSS modifications to adjust corner radii of TextField and Button

The following represents the CSS code that is needed to flatten the rounded corners on the right side of the TextField and of the left side of the Button. It also reduces the insets on the ride side of the TextField to only have one vertical line between the TextField and the Button.

```
.combined-control {

}
.combined-control:focused > .text-input {
    -fx-background-radius: 3 0 0 3, 2 0 0 2, 4 0 0 4, 0;
}
.combined-control:focused > .button {
    -fx-background-radius: 0 3 3 0, 0 2 2 0, 0 1 1 0, 0 4 4 0, 0 1 1 0;
}

.combined-control > .text-input,
.combined-control > .text-input:focused {
    -fx-background-insets  : 0, 1 0 1 1;
    -fx-background-color   : linear-gradient(to bottom,
                                derive(-fx-text-box-border, -10%),
                                -fx-text-box-border),
                             linear-gradient(from 0px 0px to 0px 5px,
                                derive(-fx-control-inner-background, -9%),
                                -fx-control-inner-background);
    -fx-background-radius  : 3 0 0 3, 2 0 0 2;
    -fx-pref-width         : 120px;
}

.combined-control > .button {
    -fx-background-radius: 0 3 3 0, 0 3 3 0, 0 2 2 0, 0 1 1 0;
    -fx-pref-width       : 36px;
}
.combined-control > .button:focused {
    -fx-background-color : -fx-outer-border, -fx-inner-border, -fx-body-color,
                           -fx-body-color;
    -fx-background-insets: 0, 1, 2, 2;
    -fx-background-radius: 0 3 3 0, 0 2 2 0, 0 1 1 0, 0 1 1 0;
}
```

Attention:

Make the control focusable
CSS modifications to create focus highlight around new control

Conclusion:

This is the most basic approach to create custom controls but nevertheless it's sometimes really useful to combine some existing controls instead of creating everything from scratch.

2.2 Extending Controls

Sometimes you just need an additional feature in an existing control and so the easiest way to get your feature is to extend an existing control.
Unfortunately this is sometimes not that easy because the JavaFX team decided to declare a lot of the control methods as final which means you can't override them in your extended control.

As an example let's take a look at the ComboBox. It is a combined control that contains a ListCell (TextField) and a StackPane (Button). So if you would like to change the behavior of the button you will discover that there is no method available in the ComboBox that gives you a handle to that button.
But there is a solution to that kind of problems you can use a trick to get a handle to that button.
We know that controls have child nodes and CSS styles applied to them. Lucky for us that there is a method called lookup which can lookup a CSS style class and return the corresponding node if it finds one.
With this trick you can get the button of a combo-box with something like follows:

```
Node arrowButton = comboBox.lookup("#arrow-button");
```

But you have to keep in mind that this only works after the CSS was applied to the control which means after the skin was instantiated. Means you can't do this in the init method of an Application class. But again there is a trick that you can use, you can add a listener to the skinProperty() of the comboBox and as soon as the listener is triggered you can run your code.

On the next page you will find some code that will explain it:

Lookup example:

```
public class Test extends Application {
    private ComboBox<String> comboBox;

    @Override public void init() {
        // Instantiate the ComboBox
        comboBox = new ComboBox<>();
        comboBox.getItems().addAll("Han Solo", "Luke Skywalker", "Darth Vader");

        // Hook up a listener to the SkinProperty of the ComboBox
        comboBox.skinProperty().addListener(o -> {
            if (null == comboBox.getSkin()) return;

            // Get Node with the style "#arrow-button"
            Node arrowButton = comboBox.lookup("#arrow-button");

            // Add an EventHandler that will print out something when pressed
            arrowButton.setOnMousePressed(e -> System.out.println("Button pressed"));
        });
    }

    @Override public void start(Stage stage) {
        StackPane pane = new StackPane(comboBox);
        pane.setPadding(new Insets(20));

        Scene scene = new Scene(pane);

        stage.setTitle("Lookup Example");
        stage.setScene(scene);
        stage.show();
    }

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

As an example let's extend a TextField by adding a little icon of a loupe.
Here is what it looks like:

The loupe icon should only be visible if the TextField is empty and not focused.

First of all we have to create a new class that extends TextField which looks like the following:

```
public class SearchTextField extends TextField {

    // ******************** Constructors *************************************
    public SearchTextField() {
        this("");
    }
    public SearchTextField(String text) {
        super(text);
    }


    // ******************** Style related ************************************
    @Override protected Skin createDefaultSkin() {
        return new SearchTextFieldSkin(SearchTextField.this);
    }

    @Override public String getUserAgentStylesheet() {
        return SearchTextField.class
                              .getResource("search-text-field.css").toExternalForm();
    }
}
```

As you can see we just have a few methods, the most important ones are the ones in the style related topic. Here we define which Skin this control should instantiate per default and which CSS stylesheet should be loaded.

As already mentioned we also need to create a Skin class where we add the loupe icon to the control. So the SearchTextFieldSkin class will look like follows:

```
public class SearchTextFieldSkin extends TextFieldSkin {
    private Region loupe;
    private double size;

    // ******************** Constructors ************************************
    public SearchTextFieldSkin(SearchTextField control){
        super(control);

        initGraphics();
        registerListeners();
    }

    // ******************** Initialization ************************************
    private void initGraphics() {
        loupe = new Region();
        loupe.getStyleClass().add("loupe");
        loupe.setFocusTraversable(false);

        getChildren().addAll(loupe);
    }

    private void registerListeners() {
        getSkinnable().heightProperty().addListener(o ->
            handleControlPropertyChanged("RESIZE"));
        getSkinnable().focusedProperty().addListener(o ->
            handleControlPropertyChanged("FOCUSED"));
    }

    // ******************** Methods *******************************************
    @Override public void layoutChildren(double x, double y, double w, double h) {
        super.layoutChildren(x, y, w, h);
        size = loupe.getMaxWidth() < 0 ? 20.8 : loupe.getWidth();
        loupe.setTranslateX(-w * 0.5 + size * 0.25);
    }

    protected void handleControlPropertyChanged(String property) {
        if ("RESIZE".equals(property)) {
            loupe.setMaxSize(getSkinnable().getHeight() * 0.8,
                             getSkinnable().getHeight() * 0.8);
        } else if ("FOCUSED".equals(property)) {
            loupe.setVisible(!getSkinnable().isFocused() &&
                             getSkinnable().getText().isEmpty());
        }
    }
}
```

In the Skin we simply added a Region and applied the CSS style class "loupe" to it. In addition we just took care about the correct size and placment of this Region by adding two listeners. The actual positioning is done in the layoutChildren method.

Now the last thing that is missing is the CSS style sheet which looks as follows:

```
.loupe {
    -fx-background-color : rgba(80, 80, 80, 0.5);
    -fx-background-insets: 0 2 0 2;
    -fx-scale-shape      : true;
    -fx-shape            : "M 47.1211 44.9996 L 45 47.1209 C 44.4375 47.684 43.6743 48
42.8789 48 C 42.0828 48 41.3196 47.684 40.7571 47.1215 L 27.1406 33.505 C 24.4614 35.0883
21.3376 36 18 36 C 8.0588 36 0 27.9412 0 18 C 0 8.0588 8.0588 0 18 0 C 27.9412 0 36
8.0588 36 18 C 36 21.3378 35.0881 24.4612 33.5054 27.141 L 47.1211 40.7571 C 48.293
41.9286 48.293 43.8281 47.1211 44.9996 ZM 18 30 C 24.627 30 30 24.6273 30 18 C 30 11.3727
24.627 6 18 6 C 11.3723 6 6 11.3727 6 18 C 6 24.6273 11.3723 30 18 30 Z";
}
```

If you already wondered where we draw the loupe icon...here it is. The magic is done in the CSS file by setting a SVG path string to the -fx-shape variable. If this is defined for a styleable node it will be drawn on the whole area of the node.
One can adjust the size a little bit by playing around with the -fx-background-insets and -fx-padding variables but in principle the SVG path will be scaled to the size of the node and will be filled by the Paint defined in -fx-background-color.

2.3 Re-Style existing controls

The standard JavaFX controls make heavy use of CSS. So all controls are styled via CSS. The style definitions are stored in one file which is called modena.css.

You could extract this file as follows:

If you have the jar tool installed:

```
- Go to the console and type:
jar xf jfxrt.jar com/sun/javafx/scene/control/skin/modena/modena.css
```

Without the jar tool installed:

- Make a copy of jfxrt.jar
- Rename it to jfxrt.zip
- Open the zip file
- You can find the file under:

```
com/sun/javafx/scene/control/skin/modena/modena.css
```

This file is more than 3000 lines long and unfortunately it's the best documentation you can get. In addition there is also a webspage on the Oracle servers that provides a basic idea of what you can do with CSS in JavaFX, you can find it here:

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html

In addition to the control specific documentation, there are some things you have to know when working with JavaFX and CSS.
First of all, it is not completely compatible to the web CSS standard. Which means you can't simply apply some CSS code from a website and hope that it will style your JavaFX application.
To avoid misunderstandings all JavaFX related CSS selectors have the prefix "-fx-".
e.g. -fx-background-color, -fx-background-radius, -fx-padding etc.

The most important thing to know when styling JavaFX controls is how the insets, corner radii and background colors work.
So here is a little example:

Remember the Aerith demo from 2006? (https://java.net/projects/aerith/sources/svn/show)

Here is a screenshot from the buttons of that demo

Creating such buttons in Swing is not really hard but you have to know how to do it, in JavaFX it's really easy but again you have to know how to do it.
Let's have a look at the design of the button.

Draw Rounded Rectangle

116 x 35 px

Corner Radius 8 px

Overlay 2nd Rounded Rectangle

114 x 33 px

Corner Radius 7px

Overlay 3rd Rounded Rectangle

112 x 31 px

Corner Radius 6 px

Button Text black

OffsetX 1px, OffsetY 1px

Overlay Button Text white

As you can see the button is made out of three Rounded Rectangles which are filled with different linear Gradients. Each additional Rectangle is 2px smaller than the former one. With this approach we automatically create some kind of a frame and with the help of the linear Gradients we can create the 3D effect.

Let's have a look at the JavaFX CSS code that is needed to create such a design.

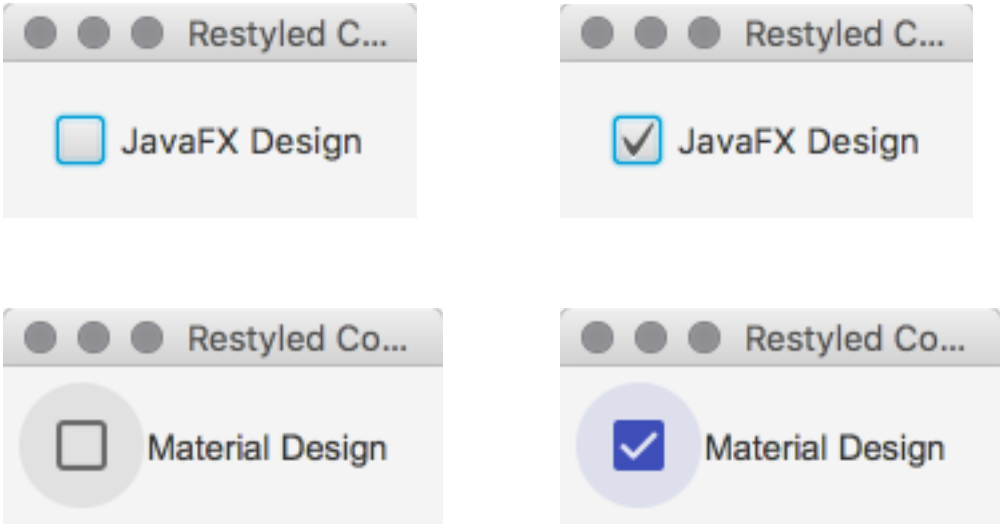
```
.my-button {
    -foreground-gradient-1: linear-gradient(to bottom, #707A8D 0%,
                                                       #959FB2 100%);
    -foreground-gradient-2: linear-gradient(to bottom, #484E59 0%,
                                                       #20252E 100%);
    -foreground-gradient-3: linear-gradient(to bottom, #AAB0BB 0%,
                                                       #7F899A 5%,
                                                       #60697A 50%,
                                                       #5B6375 51%,
                                                       #3E4658 95%,
                                                       #333A48 100%);
    -fx-pref-width       : 116px;
    -fx-pref-height      : 35px;
    -fx-background-insets: 0, 1, 2;
    -fx-background-radius: 8, 7, 6;
    -fx-background-color : -foreground-gradient-1,
                           -foreground-gradient-2,
                           -foreground-gradient-3;
}
.my-button:pressed {
    -foreground-gradient-2: linear-gradient(to bottom, #484E59 0%,
                                                       #A7AFBF 100%);
    -foreground-gradient-3: linear-gradient(to bottom, #F7D381 0%,
                                                       #F9C13E 5%,
                                                       #FEA600 50%,
                                                       #FEA200 51%,
                                                       #FD8000 95%,
                                                       #D56A00 100%);
}
.my-button .text {
    -fx-font-family: "Roboto", "sans-serif";
    -fx-font-weight: bold;
    -fx-font-size  : 20px;
    -fx-fill       : white;
    -fx-effect     : dropshadow(two-pass-box, black, 1, 0, 1, 1);
}
```

And here is the result:

The code to create such a button is fairly easy, here it is

```
public class AerithButton extends Region {
    private static final double WIDTH  = 116;
    private static final double HEIGHT = 35;
    private              Text   text;

    public AerithButton() { this(""); }
    public AerithButton(final String TEXT) {
        getStylesheets().add(
            AerithButton.class.getResource("css-basics.css").toExternalForm());
        text = new Text(TEXT);
        init();
        initGraphics();
        registerListeners();
    }

    private void init() {
        setPrefSize(WIDTH, HEIGHT);
        setMinSize(WIDTH, HEIGHT);
        setMaxSize(WIDTH, HEIGHT);
    }

    private void initGraphics() {
        getStyleClass().add("my-button");
        text.setTextOrigin(VPos.CENTER);
        text.getStyleClass().add("text");
        getChildren().setAll(text);
    }

    private void registerListeners() {
        widthProperty().addListener(o -> resize());
        heightProperty().addListener(o -> resize());
    }

    private void resize() {
        text.setX((WIDTH - text.getLayoutBounds().getWidth()) * 0.5);
        text.setY(HEIGHT * 0.5);
    }
}
```

Re-Style the JavaFX CheckBox to MaterialStyle:



To achieve this we only have to modify the existing CSS styles related to the JavaFX CheckBox and load that stylesheet to our scene.

On the next two pages you can see the CSS code that is needed to restyle the JavaFX CheckBox to make it look like a MaterialStyle CheckBox.

```
.check-box {
    -material-design-color               : #3f51b5;
    -material-design-color-transparent-12: #3f51b51f;
    -material-design-color-transparent-40: #3f51b566;

    -fx-font-family  : "Arial"; /* Roboto Regular */
    -fx-font-size    : 13px;
    -fx-label-padding: 0em 0em 0em 1.1em;
    -fx-text-fill    : -fx-text-background-color;
}
.check-box > .box {
    -fx-background-color  : transparent;
    -fx-background-insets : 0;
    -fx-border-color      : #0000008a;
    -fx-border-width      : 2px;
    -fx-border-radius     : 2px;
    -fx-padding           : 0.083333em; /* 1px */
    -fx-text-fill         : -fx-text-base-color;
    -fx-alignment         : CENTER;
    -fx-content-display   : LEFT;
}
.check-box:focused > .box {
    -fx-background-color  : #6161611f, transparent;
    -fx-background-insets : -14, 0;
    -fx-background-radius : 1024;
}
.check-box:pressed > .box {
    -fx-background-color  : -material-design-color-transparent-12,
                            transparent;
    -fx-background-insets : -14, 0;
    -fx-background-radius : 1024;
}
.check-box:selected > .box {
    -fx-background-color  : -material-design-color;
    -fx-background-radius : 2px;
    -fx-background-insets : 0;
    -fx-border-color      : transparent;
}
.check-box:selected:focused > .box {
    -fx-background-color  : -material-design-color-transparent-12,
                            -material-design-color;
    -fx-background-insets : -14, 0;
    -fx-background-radius : 1024, 2px;
    -fx-border-color      : transparent;
}
.check-box:disabled {
    -fx-opacity: 0.46;
}
```

```
.check-box > .box > .mark {
    -fx-background-color: null;
    -fx-padding         : 0.45em;
    -fx-scale-x         : 1.1;
    -fx-scale-y         : 0.8;
    -fx-shape           : "M32.49,10.915 L31.787,11.637 L29.936,13.542 L28.697,14.813
L27.315,16.232 L25.836,17.749 L24.308,19.315 L22.777,20.88 L21.294,22.395 L20.583,23.118
L19.903,23.811 L19.257,24.466 L18.653,25.078 L18.095,25.642 L17.589,26.149 L17.142,26.598
L16.759,26.978 L16.591,27.144 L16.441,27.291 L16.306,27.42 L16.188,27.533 L16.08,27.633
L15.978,27.725 L15.899,27.79 L15.776,27.885 L15.513,28.051 L14.878,28.289 L13.711,28.168
L4.781,19.393 L7.584,16.539 L14.625,23.458 L14.756,23.326 L15.256,22.825 L15.808,22.265
L16.409,21.657 L17.052,21.005 L17.729,20.315 L18.438,19.594 L19.918,18.082 L21.446,16.52
L22.973,14.956 L24.451,13.44 L25.832,12.023 L27.069,10.752 L28.92,8.849 L29.621,8.127
L32.49,10.915 z";
}
.check-box:indeterminate > .box {
    -fx-background-color  : -material-design-color-transparent-40;
    -fx-background-radius : 2px;
    -fx-background-insets : 0;
    -fx-border-color      : transparent;
}
.check-box:indeterminate  > .box > .mark {
    -fx-background-color: rgba(255, 255, 255, 0.87);
    -fx-shape           : "M0,0H10V2H0Z";
}
.check-box:selected > .box > .mark {
    -fx-background-color : rgba(255, 255, 255, 0.87);
    -fx-background-insets: 0;
}
```

2.4 Region based Controls

When you think about creating your own controls you should always start with extending a Region. The Region node is a really lightweight node that is styleable via CSS and it supports dynamic layout by sizing and positioning children during a layout pass. Another advantage of using Region is the fact that you can create a control that has only one class and it's not needed to use CSS.

Let's take a look at a simple Led control. It contains everything we need to get a better understanding of how to create Region based controls.
It contains three nodes which will be styled with CSS. The color of the Led will be styled by using a StyleableProperty and the state (on/off) will be triggered by using a CSS Pseudo Class.

If you take a closer look to this features you will figure out that we have to different aspects

- Logic (Properties, Getters, Setters etc.)
- Visuals (Resizing, Drawing, React on property changes, etc.)

The Region based controls contain both aspects where the Control+Skin based controls separate the Logic from the Visuals (more info in chapter 2.5).

Before we start let's have a quick look at CSS Pseudo Classes and Styleable Properties.

A CSS PseudoClass with the name "on" can be defined as follows:

```
private static final PseudoClass ON_PSEUDO_CLASS = PseudoClass.getPseudoClass("on");
private BooleanProperty on = new SimpleBooleanProperty(this, "on", false);
```

In the CSS Stylesheet you can now make use of it as follows:

```
.css-class {
    -fx-background-color: red;
}

.css-class:on {
    -fx-background-color: blue;
}
```

Using CSS Pseudo Classes is really powerful because you can apply CSS styles dependent on a boolean variable (in the example this is the BooleanProperty on);

A StyleableProperty of type Color that we will use define the color of our LED control.
Using StyleableProperties in JavaFX makes it possible to set properties in your control via CSS. We will use this feature to set the color of our Led control via CSS.

Because it's not that well documented here is the code that is needed for the StyleableProperty<Color>

```
// Member Variables
private static final StyleablePropertyFactory<MY_CTRL> FACTORY =
    new StyleablePropertyFactory<>(Control.getClassCssMetaData());

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

private final StyleableProperty<Color>                 color   =
    new SimpleStyleableObjectProperty<>(COLOR, this, "color");


// Getter and setter methods
public Color getColor() { return color.getValue(); }
public void setColor(final Color COLOR) { color.setValue(COLOR); }
public ObjectProperty<Color> colorProperty() {
    return (ObjectProperty<Color>) color;
}


// Style related methods
public static List<CssMetaData<? extends Styleable, ?>> getClassCssMetaData() {
    return FACTORY.getCssMetaData();
}

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

In the CSS Stylesheet you can now make use of it as follows:

```
.my-control {
    -color: blue;
}
```

Now we create a new class that extends Region and add our member variables as follows:

```
public class Led extends Region {
    // Model/Controller related
    private static final StyleablePropertyFactory<Led> FACTORY =
        new StyleablePropertyFactory<>(Region.getClassCssMetaData());
    private static final PseudoClass ON_PSEUDO_CLASS =
        PseudoClass.getPseudoClass("on");
    private BooleanProperty on;
    private static final CssMetaData<Led, Color> COLOR =
        FACTORY.createColorCssMetaData("-color", s -> s.color, Color.RED, false);
    private final StyleableProperty<Color> color;

    // View related
    private static final double PREFERRED_SIZE = 16;
    private static final double MINIMUM_SIZE   = 8;
    private static final double MAXIMUM_SIZE   = 1024;
    private double size;
    private Region frame;
    private Region main;
    private Region highlight;
    private InnerShadow innerShadow;
    private DropShadow glow;

    ...
}
```

In the constructor of our control we initialize the Properties (BooleanProperty, StyleableProperty) and call some addition methods as follows:

```
public Led() {
    on = new BooleanPropertyBase(false) {
        @Override protected void invalidated() {
            pseudoClassStateChanged(ON_PSEUDO_CLASS, get());
        }
        @Override public Object getBean() {
            return this;
        }
        @Override public String getName() {
            return "on";
        }
    };
    color = new SimpleStyleableObjectProperty<>(COLOR, this, "color");
    init();
    initGraphics();
    registerListeners();
}
```

Now we add three methods to

- apply the initial size
- setup the scene graph
- register listeners to properties of interest

The init() method:

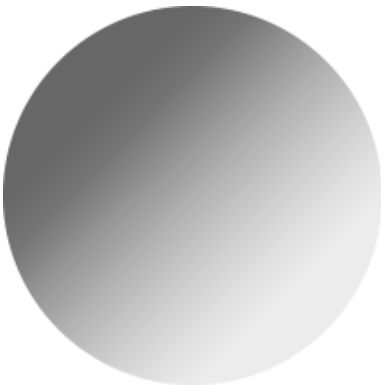
```
private void init() {
    if (Double.compare(getWidth(), 0) <= 0 ||
        Double.compare(getHeight(), 0) <= 0 ||
        Double.compare(getPrefWidth(), 0) <= 0 ||
        Double.compare(getPrefHeight(), 0) <= 0) {
            setPrefSize(PREFERRED_SIZE, PREFERRED_SIZE);
    }
    if (Double.compare(getMinWidth(), 0) <= 0 ||
        Double.compare(getMinHeight(), 0) <= 0) {
            setMinSize(MINIMUM_SIZE, MINIMUM_SIZE);
    }
    if (Double.compare(getMaxWidth(), 0) <= 0 ||
        Double.compare(getMaxHeight(), 0) <= 0) {
            setMaxSize(MAXIMUM_SIZE, MAXIMUM_SIZE);
    }
}
```

In the next method we set up the scene graph once (adding nodes to it). After that we only modify properties of these nodes and try to avoid adding/removing at runtime.

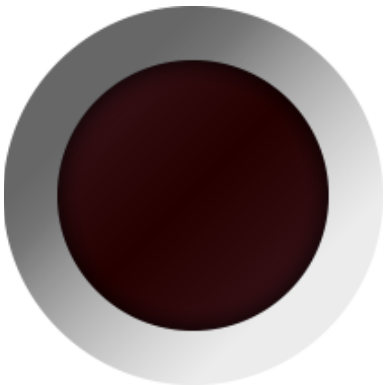
The initGraphics() method:

```
private void initGraphics() {
    // Apply the base CSS style class to the control
    getStyleClass().add("led");

    // Create the needed nodes and apply their CSS style classes
    frame = new Region();
    frame.getStyleClass().setAll("frame");

    main = new Region();
    main.getStyleClass().setAll("main");

    // We handle effects in code because we have to chain them and
    // we need to calculate the shadow spread dependent of the size
    innerShadow = new InnerShadow(BlurType.TWO_PASS_BOX, Color.rgb(0,0,0,0.65),8,0,0,0);

    glow = new DropShadow(BlurType.TWO_PASS_BOX, getColor(),20,0,0,0);
    glow.setInput(innerShadow);

    highlight = new Region();
    highlight.getStyleClass().setAll("highlight");

   // Add all nodes
   getChildren().addAll(frame, main, highlight);
}
```

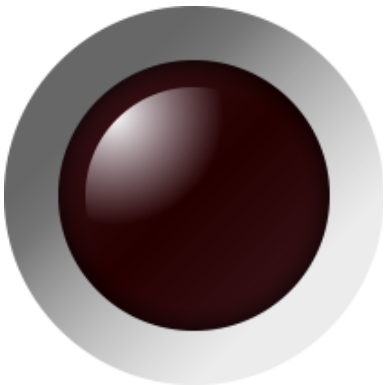
With this the visuals of the Led will be created by overlaying the three nodes as follows:



frame



frame + main



frame + main + highlight

In the next method we hookup listeners to all properties we are interested in. In our case these are

- widthProperty() to react on resizing
- heightProperty() to react on resizing
- onProperty() to toggle effects
- colorProperty() to set the Led to the right color

The registerListeners() method:

```
private void registerListeners() {
    widthProperty().addListener(o -> handleControlPropertyChanged("RESIZE"));
    heightProperty().addListener(o -> handleControlPropertyChanged("RESIZE"));
    onProperty().addListener(o -> handleControlPropertyChanged("ON"));
    colorProperty().addListener(o -> handleControlPropertyChanged("COLOR"));
}
```

As you can see we always call the handleControlPropertyChanged() method where the actual code will be placed to react on the property changes.

The handleControlPropertyChanged() method:

```
protected void handleControlPropertyChanged(final String PROPERTY) {
    if ("RESIZE".equals(PROPERTY)) {
        resize();
    } else if ("COLOR".equals(PROPERTY)) {
        main.setStyle(String.join("",
                                  "-color: ",
                                  (getColor()).toString().replace("0x", "#"), ";"));
        resize();
    } else if ("ON".equals(PROPERTY)) {
        main.setEffect(isOn() ? glow : innerShadow);
    }
}
```

The next block of methods are simply the getters and setters for our properties. It will look like follows:

```
public boolean isOn() { return on.get(); }
public void setOn(final boolean ON) { on.set(ON); }
public BooleanProperty onProperty() { return on; }


// ******************* CSS Stylable Properties ***************************
public Color getColor() { return color.getValue(); }
public void setColor(final Color COLOR) { color.setValue(COLOR); }
public ObjectProperty<Color> colorProperty() { return (ObjectProperty<Color>) color; }
```

Now we only have to add two more method blocks to our control before we will be ready to go, the first one contains the methods related to style and will look as follows:

```
// ******************* Style related ************************************
@Override public String getUserAgentStylesheet() {
    // Defines the default style sheet that belongs to this control
    return Led.class.getResource("led.css").toExternalForm();
}

public static List<CssMetaData<? extends Styleable, ?>> getClassCssMetaData() {
    // Returns the list of css meta data that is needed for StyleableProperties
    return FACTORY.getCssMetaData();
}

@Override public List<CssMetaData<? extends Styleable, ?>> getCssMetaData() {
    // Returns the list of css meta data that is needed for StyleableProperties
    return FACTORY.getCssMetaData();
}
```

Now the only thing that is missing is the resizing logic which we will add in the resize() method:

```
// ******************** Resizing ***************************************
private void resize() {
    double width  = getWidth() - getInsets().getLeft() - getInsets().getRight();
    double height = getHeight() - getInsets().getTop() - getInsets().getBottom();
    size          = width < height ? width : height;

    if (size > 0) {
        // Center component
        if (getWidth() > getHeight()) {
            setTranslateX(0.5 * (getWidth() - size));
        } else if (getHeight() > getWidth()) {
            setTranslateY(0.5 * (getHeight() - size));
        }

        // Adjust the shadow radii related to the current size
        innerShadow.setRadius(0.07 * size);
        glow.setRadius(0.36 * size);
        glow.setColor(getColor());

        // Adjust size and location of the child nodes
        frame.setPrefSize(size, size);

        main.setPrefSize(0.72 * size, 0.72 * size);
        main.relocate(0.14 * size, 0.14 * size);

        // Set effect dependent on current state
        main.setEffect(isOn() ? glow : innerShadow);

        highlight.setPrefSize(0.58 * size, 0.58 * size);
        highlight.relocate(0.21 * size, 0.21 * size);
    }
}
```

And that's all we need for our Region based custom control.

Attention:

Region doesn't offer a getChildren() method per default you won't be able to use it in SceneBuilder. But there is a solution for this:

- add @DefaultProperty("children") annotation to your class
- add getChildren() method which returns super.getChildren()

Those of you that worked with custom controls already might have seen that we don't put additional code in the layoutChildren() method. I usually try to avoid overriding this method because it will be called on every layout cycle where this control was marked dirty. For simple controls this is no problem but if you do a lot of complex drawings etc. you have add additional logic with e.g. your own dirty flags to avoid to many repaints of your control.
In my approach we have simply attached listeners to the widthProperty and heightProperty which only will be called when the size of the control changed which is most of the times good enough.
So in principle you could also put the code that is in the resize() method in the layoutChildren() method and you should see the same results.

2.5 Control + Skin based controls

There are controls that have a common model but could have different visualizations for it. If we take a look at our Led example from chapter 2.4 we see that it has one property to define it's color and another property to define a state (on/off). That's fine for a LED control but that would also apply for a simple switch control.
This means we could use the same model for another visualization. You could simply create another control that extends Region, copy the code from the LED control and modify the visualization but it would make more sense to share the model and just create another visualization for it.
To do that we take the code from the Region based Led control and split it up into two parts, the logic and the visuals.
We will put the logic in a Control class and the visuals in a Skin class as follows:

```
public class CustomControl extends Control {
    public enum SkinType { LED, SWITCH }

    private static final StyleablePropertyFactory<CustomControl> FACTORY =
        new StyleablePropertyFactory<>(Control.getClassCssMetaData());

    // CSS pseudo classes
    private static final PseudoClass ON_PSEUDO_CLASS = PseudoClass.getPseudoClass("on");
    private BooleanProperty on;

    // CSS Styleable property
    private static final CssMetaData<CustomControl, Color> COLOR =
        FACTORY.createColorCssMetaData("-color", s -> s.color, Color.RED, false);
    private final StyleableProperty<Color> color;

    // Properties
    private SkinType skinType;


    // ******************** Constructors *************************************
    public CustomControl() {
        this(SkinType.LED);
    }
    public CustomControl(final SkinType SKIN_TYPE) {
        getStyleClass().add("custom-control");
        skinType = SKIN_TYPE;
        on = new BooleanPropertyBase(false) {
            @Override protected void invalidated() {
                pseudoClassStateChanged(ON_PSEUDO_CLASS, get());
            }
            @Override public Object getBean() { return CustomControl.this; }
            @Override public String getName() { return "on"; }
```

```
        };
        color = new SimpleStyleableObjectProperty<>(COLOR, CustomControl.this, "color");
    }


    // ******************** Methods ******************************************
    public boolean isOn() { return on.get(); }
    public void setOn(final boolean ON) { on.set(ON); }
    public BooleanProperty onProperty() { return on; }


    // ******************** CSS Styleable Properties *************************
    public Color getColor() { return color.getValue(); }
    public void setColor(final Color COLOR) { color.setValue(COLOR); }
    public ObjectProperty<Color> colorProperty() {
        return (ObjectProperty<Color>) color;
    }


    // ******************** Style related ************************************
    @Override protected Skin createDefaultSkin() {
        switch(skinType) {
            case SWITCH: return new SwitchSkin(CustomControl.this);
            case LED   :
            default    : return new LedSkin(CustomControl.this);
        }
    }

    @Override public String getUserAgentStylesheet() {
        switch(skinType) {
            case SWITCH:
                return CustomControl.class.getResource("switch.css").toExternalForm();
            case LED   :
            default    :
                return CustomControl.class
                                    .getResource("custom-control.css").toExternalForm();
        }
    }

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

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

As you can see we've only added the properties of Region based LED. We also prepared already for two separate Skin's (LedSkin and SwitchSkin) and two CSS style sheets (switch.css and custom-control.css).
To be able to select the Skin we implemented an Enum SkinType that defines the Skin that should be used, the default Skin in our case is the LedSkin.
In the next step we will create the LedSkin that contains the visualization code from the Region based Led control.

```
public class LedSkin extends SkinBase<CustomControl> implements Skin<CustomControl> {
    private static final double      PREFERRED_SIZE = 16;
    private static final double      MINIMUM_SIZE   = 8;
    private static final double      MAXIMUM_SIZE   = 1024;
    private              double      size;
    private              Region      frame;
    private              Region      main;
    private              Region      highlight;
    private              InnerShadow innerShadow;
    private              DropShadow  glow;


    // ******************** Constructors **************************************
    public LedSkin(final CustomControl CONTROL) {
        super(CONTROL);
        init();
        initGraphics();
        registerListeners();
    }

    ...
}
```

As you can see the Skin extends SkinBase and implements the Skin Interface. It takes the Control as a parameter in the constructor that follows the same scheme of calling the init(), initGraphics() and registerListeners() methods.

This three methods are part of the Initialization block and will look as follows:

```
// ******************** Initialization ************************************
private void init() {
    if (Double.compare(getSkinnable().getPrefWidth(), 0.0) <= 0 ||
        Double.compare(getSkinnable().getPrefHeight(), 0.0) <= 0 ||
        Double.compare(getSkinnable().getWidth(), 0.0) <= 0 ||
        Double.compare(getSkinnable().getHeight(), 0.0) <= 0) {
            if (getSkinnable().getPrefWidth() > 0 && getSkinnable().getPrefHeight() > 0){
                getSkinnable().setPrefSize(getSkinnable().getPrefWidth(),
                getSkinnable().getPrefHeight());
            } else {
                getSkinnable().setPrefSize(PREFERRED_SIZE, PREFERRED_SIZE);
            }
    }
    if (Double.compare(getSkinnable().getMinWidth(), 0.0) <= 0 ||
        Double.compare(getSkinnable().getMinHeight(), 0.0) <= 0) {
            getSkinnable().setMinSize(MINIMUM_SIZE, MINIMUM_SIZE);
    }
    if (Double.compare(getSkinnable().getMaxWidth(), 0.0) <= 0 ||
        Double.compare(getSkinnable().getMaxHeight(), 0.0) <= 0) {
            getSkinnable().setMaxSize(MAXIMUM_SIZE, MAXIMUM_SIZE);
    }
}

private void initGraphics() {
    frame = new Region();
    frame.getStyleClass().setAll("frame");

    main = new Region();
    main.getStyleClass().setAll("main");
    main.setStyle(String.join("",
                              "-color: ",
                              getSkinnable().getColor().toString().replace("0x", "#"),
                              ";"));

    innerShadow = new InnerShadow(BlurType.TWO_PASS_BOX, Color.rgb(0,0,0,0.65),8,0,0,0);

    glow = new DropShadow(BlurType.TWO_PASS_BOX, getSkinnable().getColor(),20,0,0,0);
    glow.setInput(innerShadow);

    highlight = new Region();
    highlight.getStyleClass().setAll("highlight");

    getChildren().addAll(frame, main, highlight);
}
```

```
private void registerListeners() {
    getSkinnable().widthProperty().addListener(o ->
        handleControlPropertyChanged("RESIZE") );
    getSkinnable().heightProperty().addListener(o ->
        handleControlPropertyChanged("RESIZE") );
    getSkinnable().colorProperty().addListener(o ->
        handleControlPropertyChanged("COLOR"));
    getSkinnable().onProperty().addListener(o ->
        handleControlPropertyChanged("ON") );
}
```

In principle this is the same code as in the Region based control. The only difference is that you have to call getSkinnable() to get access to the Control class methods where in the Region you can simply call the methods you need.
The methods block only contains the handleControlPropertyChanged method because all the properties are handled in the Control class. And again this method looks nearly the same as in the Region based Led control:

```
// ******************** Methods ********************************************
protected void handleControlPropertyChanged(final String PROPERTY) {
    if ("RESIZE".equals(PROPERTY)) {
        resize();
    } else if ("COLOR".equals(PROPERTY)) {
        main.setStyle(String.join("",
                                  "-color: ",
                                  getSkinnable().getColor()
                                                .toString().replace("0x", "#"), ";"));
        resize();
    } else if ("ON".equals(PROPERTY)) {
        main.setEffect(getSkinnable().isOn() ? glow : innerShadow);
    }
}
```

The last thing we need to add is the resizing block. For a simple control like the LED it would be no problem to put this code in the layoutChildren method but to stay on the scheme we use the resize() method here:

```
// ******************** Resizing ************************************
private void resize() {
    double width  = getSkinnable().getWidth() -
                    getSkinnable().getInsets().getLeft() -
                    getSkinnable().getInsets().getRight();
    double height = getSkinnable().getHeight() -
                    getSkinnable().getInsets().getTop() -
                    getSkinnable().getInsets().getBottom();
    size          = width < height ? width : height;

    if (size > 0) {
        innerShadow.setRadius(0.07 * size);
        glow.setRadius(0.36 * size);
        glow.setColor(getSkinnable().getColor());

        frame.setMaxSize(size, size);

        main.setMaxSize(0.72 * size, 0.72 * size);
        main.relocate(0.14 * size, 0.14 * size);
        main.setEffect(getSkinnable().isOn() ? glow : innerShadow);

        highlight.setMaxSize(0.58 * size, 0.58 * size);
        highlight.relocate(0.21 * size, 0.21 * size);
    }
}
```

The CSS style sheet is again more or less the same as the led.css of the Region based control. The only difference is that we name the base style class .custom-control instead of .led.

Just to be complete here is the custom-control.css file:

```
.custom-control {
    -color: red;
}

.custom-control .frame {
    -fx-background-color : linear-gradient(from 14% 14% to 84% 84%,
                                           rgba(20, 20, 20, 0.64706) 0%,
                                           rgba(20, 20, 20, 0.64706) 15%,
                                           rgba(41, 41, 41, 0.64706) 26%,
                                           rgba(200, 200, 200, 0.40631) 85%,
                                           rgba(200, 200, 200, 0.3451) 100%);
    -fx-background-radius: 1024px;
}
.custom-control .main {
    -fx-background-color : linear-gradient(from 15% 15% to 83% 83%,
                                           derive(-color, -80%) 0%,
                                           derive(-color, -87%) 49%,
                                           derive(-color, -80%) 100%);
    -fx-background-radius: 1024px;
}
.custom-control:on .main {
    -fx-background-color: linear-gradient(from 15% 15% to 83% 83%,
                                          derive(-color, -23%) 0%,
                                          derive(-color, -50%) 49%,
                                          -color 100%);
}
.custom-control .highlight {
    -fx-background-color : radial-gradient(center 15% 15%, radius 50%, white 0%,
transparent 100%);
    -fx-background-radius: 1024;
}
```

And that's all we need for the LedSkin, when we use our CustomControl without any parameters it will look like follows (left default, right on == true):

Now that we have finished the LedSkin let's have a look at the SwitchSkin. Again this Skin will be really simple and in this case we will only use CSS to style it. The idea is to build something like follows:

In principle it contains two Regions (background and thumb) that we will style by using CSS. In this case I don't want the control to be resizeable. Therefor I simply set the min-, max- and preferredSize to the same values which prevents the Regions from being resized by the layout container.
That makes the Skin even simpler than the LedSkin, so let's start with the implementation by creating the SwitchSkin class that also extends SkinBase and implements the Skin interface as follows:

```
public class SwitchSkin extends SkinBase<CustomControl> implements Skin<CustomControl> {
    private static final double             PREFERRED_WIDTH  = 76;
    private static final double             PREFERRED_HEIGHT = 46;
    private              Region             switchBackground;
    private              Region             thumb;
    private              Pane               pane;
    private              TranslateTransition translate;


    // ******************** Constructors **************************************
    public SwitchSkin(final CustomControl CONTROL) {
        super(CONTROL);
        initGraphics();
        registerListeners();
    }

    ...
}
```

As you can see we have just a few member variables and here we also do not need an init() method because we will fix the size of the control anyway.

The Initialization block now only contains the initGraphics() and registerListeners() method and will look as follows:

```
// ******************** Initialization ************************************
private void initGraphics() {
    switchBackground = new Region();
    switchBackground.getStyleClass().add("switch-background");
    switchBackground.setStyle(String.join("",
                                          "-color: ",
                                          getSkinnable().getColor().toString()
                                                        .replace("0x", "#"),
                                          ";"));

    thumb = new Region();
    thumb.getStyleClass().add("thumb");
    if (getSkinnable().isOn()) { thumb.setTranslateX(32); }

    translate = new TranslateTransition(Duration.millis(70), thumb);

    pane = new Pane(switchBackground, thumb);
    getChildren().add(pane);
}

private void registerListeners() {
    getSkinnable().colorProperty().addListener(o ->
        handleControlPropertyChanged("COLOR"));
    getSkinnable().onProperty().addListener(o -> handleControlPropertyChanged("ON") );
    thumb.setOnMousePressed(e -> getSkinnable().setOn(!getSkinnable().isOn()));
}
```

Here we added the two Regions (switchBackground and thumb) to an additional Pane which we then add to the Control. The additional Pane is used to be able to place the child nodes as we need them to be where the enclosing Pane will be resized by the layout container.

Now we just need to add the Methods block and we are ready to go, so here it is:

```
// ******************** Methods ******************************************
@Override public void layoutChildren(double x, double y, double w, double h) {
    super.layoutChildren(x, y, w, h);
    switchBackground.relocate((w - PREFERRED_WIDTH) * 0.5, (h - PREFERRED_HEIGHT) * 0.5);
    thumb.relocate((w - PREFERRED_WIDTH) * 0.5, (h - PREFERRED_HEIGHT) * 0.5);
}

protected void handleControlPropertyChanged(final String PROPERTY) {
    if ("COLOR".equals(PROPERTY)) {
        switchBackground.setStyle(String.join("",
                                              "-color: ",
                                              getSkinnable().getColor().toString()
                                                            .replace("0x", "#"),
                                              ";"));
    } else if ("ON".equals(PROPERTY)) {
        if (getSkinnable().isOn()) {
            // move thumb to the right
            translate.setFromX(2);
            translate.setToX(32);
        } else {
            // move thumb to the left
            translate.setFromX(32);
            translate.setToX(2);
        }
        translate.play();
    }
}
```

In contrast to the LedSkin we added some interaction here by adding an EventListener<MouseEvent> to the thumb in the registerListeners() method above. With this we can click the thumb which will toggle the control on and off.
By hooking up a listener to the onProperty() of the Control class we can react on that change by translating the thumb in x-direction.

And that's all the code we need to create the SwitchSkin now the only thing that's missing is again the CSS style sheet. So here it is:

```
.custom-control {
    -color: #4bd865;
}

.custom-control .switch-background {
    -fx-pref-width       : 76;
    -fx-pref-height      : 46;
    -fx-min-width        : 76;
    -fx-min-height       : 46;
    -fx-max-width        : 76;
    -fx-max-height       : 46;
    -fx-background-radius: 1024;
    -fx-background-color : #a3a4a6;
}
.custom-control:on .switch-background {
    -fx-background-radius: 1024;
    -fx-background-color : -color;
}
.custom-control .thumb {
    -fx-translate-x      : 2;
    -fx-translate-y      : 2;
    -fx-pref-width       : 42;
    -fx-pref-height      : 42;
    -fx-min-width        : 42;
    -fx-min-height       : 42;
    -fx-max-width        : 42;
    -fx-max-height       : 42;
    -fx-background-radius: 1024;
    -fx-background-color : white;
    -fx-effect           : dropshadow(two-pass-box, rgba(0, 0, 0, 0.3), 1, 0.0, 0, 1);
}
```

And if we instantiate the CustomControl with the SkinType.SWITCH we will get the following: