



Mastering **JavaFX 8** Controls

Create Custom JavaFX Controls for
Cross-Platform Applications



Hendrik Ebbers

Oracle
Press™

As a next step, say you want to style the `PieChart` with CSS. You create a new style sheet and apply it to the scene graph, as shown here:



```
 .default-color0.chart-pie { -fx-pie-color: blue; }
.default-color1.chart-pie { -fx-pie-color: lightblue; }
.default-color2.chart-pie { -fx-pie-color: darkblue; }
.chart-pie-label-line {
    -fx-stroke: black;
}
.chart-pie-label {
    -fx-fill: black;
    -fx-font-size: 0.7em;
}
.chart-legend {
    -fx-background-color: lightyellow;
}
```

Figure 9-8 shows the result on the bottom. In the style sheet, you will find an additional feature of the CSS support in JavaFX. When styling charts, JavaFX will automatically create CSS classes for all the data parts of the chart. In the example, the classes `default-color0`, `default-color1`, and `default-color3` can be used to style each of the data sections of the pie chart.

Best Practices for Styling Applications and Controls

When styling an application, it is important not to mix too many CSS style sheets. In most cases, a single CSS file will fit all the needs of an application. Because *Modena* is defined as the default user agent style sheet in JavaFX, a style sheet for an application must contain only the changes to *Modena*.

First, apply all global rules. Let's assume the application must use a special skin for buttons. In this case, a rule that uses the `.button` pseudoclass should be defined. Here, it is important that all the different states of the control will be tested. For the button type, for example, developers will often need to change the style for states, such as `hover`, `pressed`, and `focused`, so CSS pseudoclasses should be used. If this isn't done, a user gets no visual feedback when clicking a button. The following example defines how the needed rules for a new button style may look:

```
 .toggle-button, .button{
    -fx-background-color: red;
    -fx-background-insets: 0.0;
    -fx-background-radius: 2.0;
    -fx-border-width: 0.0;
    -fx-padding: 6;
    -fx-text-fill: black;
    -fx-alignment: CENTER;
    -fx-content-display: LEFT;
}


.toggle-button:focused,
.button:focused,
.button:default:focused {
    -fx-background-color: lightcoral;
}
```

```
.toggle-button:focus:selected {
    -fx-background-color: lightcoral, lightcyan;
    -fx-background-insets: 0.0, 0.2em;
}


.toggle-button:armed,
.toggle-button:selected,
.button:armed,
.button:default:armed {
    -fx-background-color: darkred;
}
```

As you can see, the selectors of the rules contain comma-separated lists. By doing this, only a few rules are needed to specify all the needed styling information for a `Button` and a `ToggleButton` in the different states.

As a next step, define special types of the controls. In this case, it is best to define custom CSS classes. Let's say all buttons in menus should have a smaller font. Here, an `app-menu-button` CSS class is defined that contains a value for the `-fx-font-size` property:


```
 .app-menu-button {
    -fx-font-size: 8pt;
}
```

If the application contains some special buttons that are used in only one view, you can define these controls with a unique ID. In this case, the ID selector of CSS should be used:


```
 #shut-down-button {
    -fx-font-size: 16pt;
}
```

Doing this for all the needed control types defines a clear hierarchy of styling.

To create an even better overview of the CSS styling and create a more refactorable style sheet, you should use the `root` style class. The `root` class is applied to the `root` node of the scene graph, and properties that are defined in the `root` style can be used in any other CSS rule. If, for example, you define a style for an application that has blue as its main color, you can define the `root` style class as shown in the following snippet:

```
 .root {
    -fx-color-base: blue;
}
```


Here, the property `-fx-color-base` is defined in the `root`. The big benefit of this is that the property can be reused in each CSS rule of the style sheet. The following code shows another abstract of the application CSS:

```
 .button{
    -fx-background-color: -fx-color-base;
}

.context-menu .label {
    -fx-text-fill: -fx-color-base;
}
```

The `-fx-color-base` property is reused in two rules here. Once you have extracted all the basic values as global properties to the `root` rule of a style sheet, it is easy to refactor the styling by changing only one value. Another benefit is that you automatically provide more consistency in the UI because the same values will be used in a lot of control types.

When looking at the Modena CSS file, you will see that a lot of properties are defined in the `root` section of the style sheet. You can reuse these properties in custom application style sheets or define new values for them. By doing this, you can change the complete style of an application with only a few lines of CSS, as shown here:

```
 .root{
    -fx-font-size: 14pt;
    -fx-font-family: "Verdana";
    -fx-base: orange;
    -fx-background: yellow;
}
```

A custom style sheet that contains only these lines will change the complete look of an application. Figure 9-9 shows the sample application that is styled by this style sheet.



NOTE

In theory, an application can be styled by multiple style sheets. A style sheet will be added to a scene graph by calling `myScene.getStylesheets().add(...)`. The `getStylesheets()` method returns a `List`, and therefore it can hold and manage several style sheets. Suppose you have a default style for all your applications that is wrapped in one default style sheet. If one application needs some additional styles for specific components or the customer needs a special color for all button backgrounds, these features could be implemented in a separate style sheet. In this case, the order of the CSS style sheets in the list defines its priority. When possible, though, it is a best practice to define the styling of an application in a single style sheet.

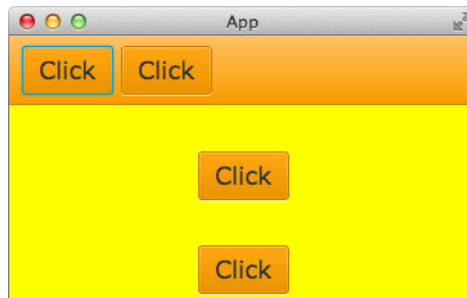




FIGURE 9-9. A styled application


A final best practice for developers creating huge CSS style sheets is the proper use of the `derive(...)` and `ladder(...)` functions. These two functions can be used to define color values in CSS. When thinking about a style for a complete application, you will mostly use some basic colors that specify the color theme of the application. In general, these colors will be used in several modifications in the UI of the application. For instance, maybe the basic color for an application is a light blue. In this case, the background of all buttons will be defined as `lightblue`, and the border of all buttons should appear in a darker blue. In this case, the `derive(...)` function can be used to derive the border color from the base color. To do this, the `derive(...)` function takes a color and computes a brighter or darker version of that color. The second parameter of the `derive(...)` function is the brightness offset that can be defined in a range from `-100%` to `100%`. You can use the `derive(...)` function as a value for a CSS property, as shown in the following snippet:

```
 .root{
    -fx-base: lightblue;
    -fx-border-base: derive(-fx-base, -50%);
    -fx-focus-base: derive(-fx-base, 50%);
}
```

In this case, only the value of the `-fx-base` property needs to be changed to affect all defined colors. In addition to the `derive(...)` function, the `ladder(...)` function can be used to interpolate between colors. Here, a gradient must be specified, and the brightness of the provided color parameter is used to look up a color value within that gradient. The calculated color depends on the brightness of the passed color. At 0 percent brightness, the returned color will be the start color of the gradient. At 100 percent brightness, the color at the 1.0 end of the gradient is used. Here's an example:

```
 #dark-button{
    -fx-background-color: darkred;
    -fx-text-fill: ladder(darkred, white 0%, black 100%);
}
#light-button{
    -fx-background-color: lightyellow;
    -fx-text-fill: ladder(lightyellow, white 0%, black 100%);
}
```

In the CSS style sheet, the `ladder(...)` function is used to define the text color for two buttons. You create these buttons in a JavaFX application without defining any additional parameters next to their IDs:

```
 ..
Button darkButton = new Button("Dark");
darkButton.textFId("dark-button");
Button lightButton = new Button("Light");
lightButton.setId("light-button");
...
```

When running the application, the two buttons will have different text colors. The color of the text fits perfectly to the light and dark backgrounds of the buttons, as shown in Figure 9-10.

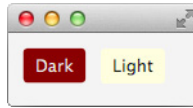


FIGURE 9-10. *Result of the `ladder()` function*

By using the brightness of the background color, the `ladder(...)` function calculates the perfect color for the text.

With the two methods shown here, it is easy to define global base colors in CSS and use derived colors in specific rules.

An Interview with Claudine Zillmann, software developer at maredit GmbH

Hi, Claudine. Many JavaFX developers know you as a CSS expert for JavaFX styling and the maintainer of AquaFX. Therefore, I think you can add some useful information and tips about styling to the more general information about CSS mentioned in this book. But before we start, can you please introduce yourself and how you came to JavaFX and CSS?

Hello, Hendrik, it's a pleasure to contribute to your work with an interview and share tips and knowledge on this topic.

Right now, I work at maredit GmbH as one of three lead developers for e-commerce projects and specialize in developing controls with our recently created web framework.

But to introduce myself, I take a step further back in time.

I will start with my first steps in styling and HTML. At the beginning, I tried some things just for fun, using 1x1-pixel images or marquee tags. Soon, school projects made me concentrate on using CSS and what it stands for. You find out a lot of benefits and concepts, such as the box model and separation of markup and styling in general. But I realized that those components were not comfortable. Especially proper usage of positioning made me mad at times.

During my studies at university, I concentrated on Java development and started to work in a company with a client-server application, specializing on development of the Swing client for seven years. In that time, JavaFX came up and became a successor of Swing. Soon, the idea of AquaFX was born at the completion of my degree. It was like, "Hey, I think JavaFX really rocks, and I want a deep dive in that technology. JavaFX is young and could need contributions. Let's create a skin!" Since that time, JavaFX has been my hobby and my favorite UI framework.

It sounds to me as you played with most of the different layout solutions that appeared over the last few years, beginning with a 1-pixel blank.gif to the CSS styling that we have today with Bootstrap and JavaFX CSS styling. Can you tell me what are the biggest differences between CSS styling for web content and JavaFX content?

The differences are not really that big, on the one hand. On the other hand, they are immense (but control-based web design comes up more and more). In an exaggerated way, I think I was lucky by not getting too serious in web design, so I do not miss a lot of stuff that might be possible with web CSS. JavaFX adapted the CSS standard and adjusted it for its UI controls. So, you have nodes (a box model), which can be styled with colors, borders, padding, effects, and so on, and which also can inherit styles by parent nodes, as known from conventional web design. If you know those concepts and the possibilities that CSS offers, it is not hard to style JavaFX applications. You just have to be aware of some tiny things:

- The naming is different: not `background-color` but `-fx-background-color`.
- You are within a control, especially when it's about positioning. This is the part where you should not forget about the concept of layout managers in UI frameworks.
- Not everything is possible with JavaFX CSS, but almost. Just stick to the reference guide before you start. That will help.
- Things that might seem impossible are not really impossible because you also might realize them in a programmatic way.

All in all, it is pretty similar. Like in web-based design, all those possibilities can be used in a good and a “less good” way. This could cause ugly code, unwanted visual results, or performance issues.

In conclusion, JavaFX combines the strength of UI frameworks with web design, so cool results can be achieved straightforwardly, even if you do not know CSS very well or at all.

Let me ask a last general CSS question before we talk about CSS for JavaFX and your AquaFX project. There are different ways you can apply CSS: by using several style sheets, defining all CSS rules in one big style sheet, or adding all styles inline, for example. Do you have any tips or a best-practice workflow for how developers should organize their CSS styles?

There is no general best-practice workflow that fits every need. I think the answer is, “It depends.” You have to ask yourself a couple of questions to find the proper organization of styles for your project. For example:

- Do I want to reuse styles?
- Can some characteristics be generalized?
- Should colors/sizes vary? Or are there other variations?

So, define the individual requirements of your own project, and then you can decide how to realize it.

Generally, if you just want to try something or modify some tiny things, inline styles might be enough. I'm not a friend of that technique at all, though.

If you create a set of your own controls, try to put all general definitions in one style sheet and add individual styles for each control. For a complete skin, I used one style sheet because you touch every single control.

(Continued)

Thanks for this hint. Let's change the focus and talk about JavaFX in combination with CSS. Why do you think each JavaFX developer should know CSS and use it when developing JavaFX applications?

This answer is pretty simple: because it makes development so much easier. The times of programmatic styling and drawing each component are over. Formatting, colors, and style information shouldn't live within your code. Why not use proven concepts of web development and apply them? Your application gets much more flexible as well. On top of that, CSS is no mystery and not that hard to learn. Does it need more reasons?

The CSS support in JavaFX is based on CSS 2.1 and adds some useful functions like `derive(...)` and `ladder(...)`. These functions are normally not part of CSS and help a developer to create flexible and more readable CSS definitions. Do you have some other hints how a developer can create readable and flexible CSS rules?

There are several hints to achieve readable and flexible CSS definitions. I think the five most important tips are

- Get to know all possibilities JavaFX already ships with by default. Take more than one look at the JavaFX CSS Reference Guide. It tells you about all those cool functions JavaFX offers for CSS.
- The other important thing to address is the source of the Modena skin. When you look at the CSS shipped with JavaFX, there are no more mysteries about the usage of CSS.
- Make your CSS flexible by learning how to use selectors properly. The smarter your selection is, the more efficient the parsing of the scene graph is.
- When it comes to colors, use looked-up colors. Looked-up colors enable a global definition of colors in some sort of variable. This then can be used in the whole CSS and, for example, altered by the functions `derive(...)` and `ladder(...)` you mentioned. With that approach, you can use a few color definitions that are held centrally and avoid a complete revision of all CSS definitions when changing the whole color scheme. This concept could even be widened by implementing your own functions. As I mentioned, almost everything is possible with JavaFX.

Many web developers started to use a dynamic style sheet language like LESS to optimize their CSS style sheets. Do you know whether workflows like this can be used in JavaFX, too?

Oh, well, actually Tom Schindl has already experimented with LESS and JavaFX. He published a little blog post about his thoughts and experiments in this promising area. If you want to know more, read it at <http://tomsondev.bestsolution.at/2013/08/07/using-less-in-javafx/>. This trend is really noteworthy.

Let's talk about AquaFX. With this project, you created a complete skin for all the basic JavaFX controls. Can you share some of the pitfalls with that you were confronted with and experiences that you earned while developing the theme?

Sure, I can. But where to start? I think, with the concept. One of the first questions I asked myself was, do I base the skin on an existing JavaFX skin, or shall I create it independently? Since AquaFX will explicitly be the look and feel of OS X for JavaFX, it couldn't be dependent

on future changes in the base skin. So, adjusting the base skin was no option, which means that every control had to be styled from scratch.

When it comes to styling controls, this is pure work and good eyes. One pitfall can be the vertical alignment of text. If you work a lot with padding, check your new controls next to each other. Some other tips for all kinds of questions can be found on my slides, loaded up on SlideShare: <http://de.slideshare.net/ClaudineZillmann/lets-get-wetbestpracticesforskinningjavafxcontrols>.

Thank you very much for this interview. Let me ask one last question: Are you planning to create some other JavaFX themes in the future?

Thank you very much for conducting this interview. It is always a pleasure. To your last question: Well, yes, the next skin, FlatterFX, is still in progress and will be published when it is finished. I guess the next steps are maybe some experiments with LESS and some optimization of AquaFX. After that, we will see what the community might ask for.

Summary

As you saw in this chapter, the CSS support in JavaFX is detailed and could therefore warrant a book of its own. This chapter covered the basics, so you should be able to create style sheets based on these practices and features. It is also important to take a deeper look at the CSS documentation of JavaFX as you begin working with CSS; all the classes, pseudoclasses, and properties are described in the documentation.

By using CSS to style a JavaFX application, you can create a perfect separation between the styling and the logic of an application. By using CSS and FXML in combination, only the business and controller logic of an application must be implemented in Java. All the layout and styling topics can be defined in languages that fit your needs. If you've already used CSS for web development, you should have no trouble becoming familiar with the functions covered in this chapter. However, the benefits of using CSS are so important that everyone should use it. Your application will be more structured, and changes in the style of an application can be defined quickly if the CSS style sheet of the application is well structured.