Chapter 12

Skinning Your Application with CSS

In This Chapter

- Applying styles in several ways
- Creating your own style sheet
- Controlling fonts through font styles
- Creating fills and borders with CSS

One of the most powerful features of JavaFX is its ability to use CSS (which stands for *Cascading Style Sheets*) to control the visual appearance of your user interface. With CSS, you can change the look and feel of your application without actually changing any of the Java code that powers your application. CSS essentially disconnects the visual aspects of your program from the application logic.

The terms *theme* and *skin* are used somewhat interchangeably to refer to the look and feel of an application. A theme or skin governs many aspects of visual appearance, including the font used for text, background fills, border styles and colors, how items react when the mouse is hovered over them, and many more.

In this chapter, I first discuss how to switch an entire application between two of the default themes provided with JavaFX. Then, you discover how to craft your own style sheets and apply them to your scenes.

Using Default Style Sheets

JavaFX comes with two built-in themes: Modena and Caspian. Modena is a new theme that was introduced with JavaFX 8; Caspian is an older theme that was used with previous versions of JavaFX.

Figure 12-1 shows a version of the Pizza Order application that I present in Chapter 11; it includes a pair of radio buttons to allow the user to switch between the Modena and Caspian themes. The window on the left side of the figure shows the Modena theme; the Caspian theme is shown on the right.

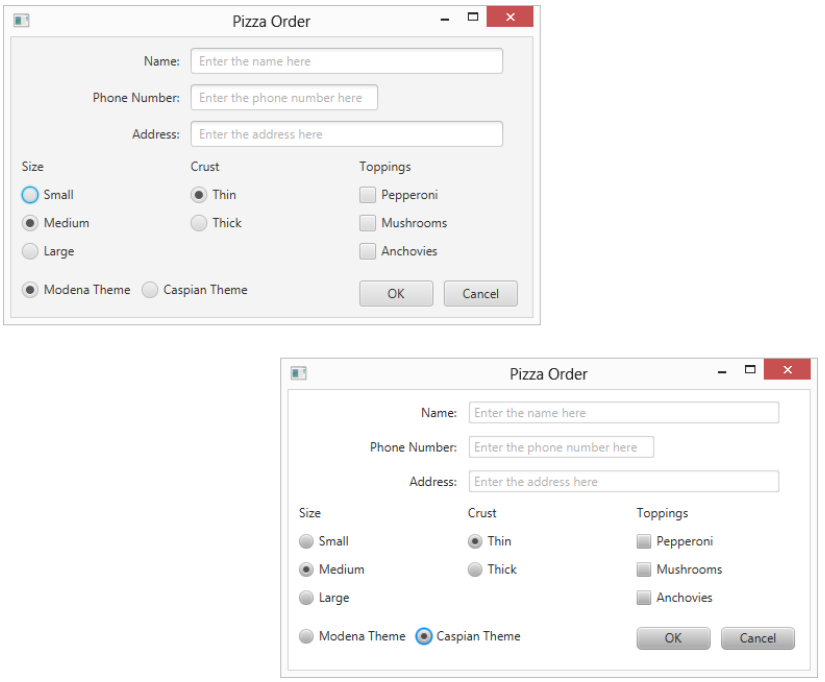

Figure 12-1: A JavaFX application shown in the Modena and Caspian themes.

To switch the theme of an application, use the `setUserAgentStylesheet` method of the `Application` class. The `Application` class defines two static fields that you can use to reference the built-in styles: `STYLESHEET_MODENA` and `STYLESHEET_CASPIAN`. Thus, to set Caspian as the style sheet, use this statement:

```
Application.setUserAgentStylesheet(
    STYLESHEET_CASPIAN);
```

Because the program used to create the screens shown in Figure 12-1 is long and identical to the Pizza Order application presented in Listing 11-2 in Chapter 11, I don't duplicate it in its entirety. The only significant addition to the Pizza Order application is the code that defines the two radio buttons at the bottom-left corner of the scene:

```
ToggleGroup groupTheme = new ToggleGroup();

RadioButton rdoModena =
    new RadioButton("Modena Theme");
rdoModena.setToggleGroup(groupTheme);
rdoModena.setSelected(true);
rdoModena.setOnAction(e ->
    {
        setUserAgentStylesheet(STYLESHEET_MODENA);
    });

RadioButton rdoCaspian =
    new RadioButton("Caspian Theme");
rdoCaspian.setToggleGroup(groupTheme);
rdoCaspian.setOnAction(e ->
    {
        setUserAgentStylesheet(STYLESHEET_CASPIAN);
    });

HBox paneTheme = new HBox(10, rdoModena, rdoCaspian);
```

As you can see, this code creates two radio buttons whose action event handlers set the theme to either Modena or Caspian. These radio buttons are added to a `ToggleGroup` and to an `HBox`. Later in the program, the `HBox` is added to a `GridPane` layout to display in the bottom-left corner of the screen.

Adding a Style Sheet to a Scene

If you want to, you can create a style sheet to replace the Modena or Caspian themes with your own theme, creating an entirely different look and feel for your application. Then, you can apply your style sheet as the application's default style sheet using `Application.setUserAgentStylesheet`, as I describe in the preceding section.

However, creating a completely new theme to apply application-wide can be a difficult task, as your style sheet must provide style information for every possible formattable node element. Instead, you may want to start by creating a smaller style sheet that just provides formatting information for the specific needs of your application. Then, you can apply the style sheet to a specific scene or to an individual node within a scene.

When you apply a style sheet to a scene, any styles contained in that style sheet override any corresponding styles in the application's default style sheet. Similarly, if you apply a style sheet to a specific node, the styles in that style sheet override any corresponding styles in the style sheet applied to the scene.

You can create as many style sheets as you want, applying different style sheets to different parent nodes within the scene. However, you'll find it easier to manage your styles and create consistency if you stick to just one style sheet applied at the scene level.

A style sheet applied to a scene or parent node is actually a separate file with the extension `.css`. The style sheet contains formatting rules that provide the specifics for the formatting you want applied to your application.

You can read about the details of creating a style sheet in the section "Creating a Style Sheet" later in this chapter. For now, I introduce a very simple style sheet named `Simple.css` that specifies the font to use for text and a background color. The `Simple.css` style sheet consists of the following lines:

```
.root
{
    -fx-background-color: lightgray;
    -fx-font-family: "serif";
    -fx-font-size: 12pt;
}
```

The first line specifies that the formatting between the curly braces that follow applies through the entire scene graph. Then, within the curly braces, three formatting rules are used to specify that the background color should be `lightgray`, the font should be `serif`, and the font size should be 12 points.

The easiest way to add a style sheet to a scene is to get the scene's style sheet collection (a scene can have more than one style sheet), use the `add` method to add the style sheet, like this:

```
scene.getStylesheets().add("Simple.css");
```

To keep the code examples in this book simple, the rest of the examples in this chapter use the simple technique shown in the preceding example. However, simply specifying the stylesheet name as the parameter to the `add` method code will work only if the style sheet file resides in the same folder as the application's class file. If the style sheet resides elsewhere, use the following code instead:

```
scene.getStylesheets().add(
    getClass().getResource("Simple.css")
                    .toExternalForm());
```

Instead of simply providing the name of the style sheet as a string, this technique calls the `getClass` method of the `Object` class, which returns a reference to the application's class. Then, it calls the `Class` `getResource` method, which accepts a string parameter that names an external resource (such as a file) that's located on the application's class path. This returns the URL of the `Simple.css` file. Finally, the `toExternalForm` method massages the URL into a form acceptable to the `getStylesheets.add` method.

The window in the top-left part of Figure 12-2 shows a JavaFX application with the `Simple.css` added to the scene. For comparison, the figure also shows the application without the `Simple.css` file. As you can see, the style sheet has changed the background color to a darker shade of gray and changed the font to a serif-style font (on Windows computers, Times New Roman is used).

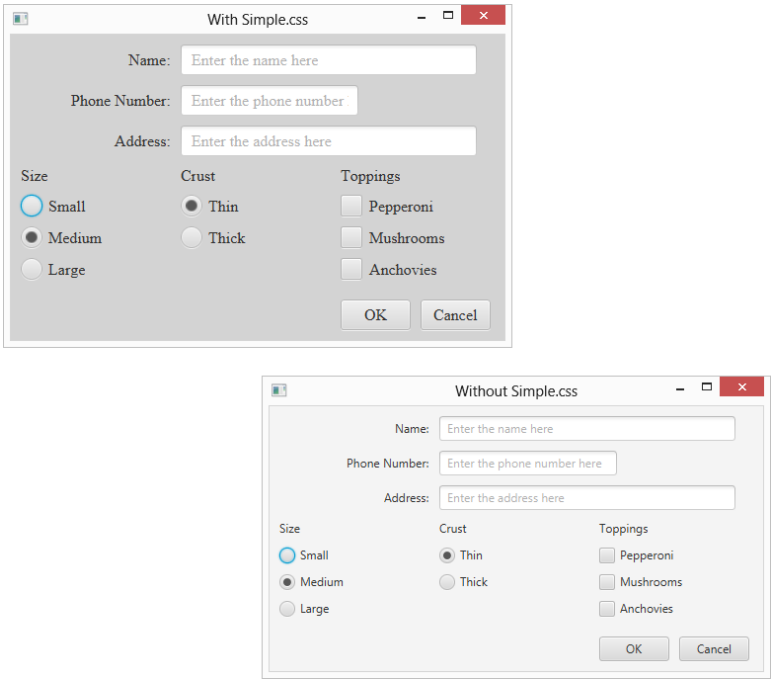

Figure 12-2: A JavaFX application with and without the `Simple.css` file style sheet.

Using Inline Styling

In addition to using separate `.css` style sheet files, JavaFX lets you apply style rules directly to any node in a scene graph by calling the node's `setStyle` method, passing the formatting rule as a string argument. For example, the following example sets the font size for a button to 15 points:

```
Button btnOK = new Button("OK");
btnOK.setStyle("-fx-font-size: 15pt");
```

As a general rule, I recommend against using inline styles except for unusual situations. That's because inline styles make it more difficult to change the formatting of your application's user interface. Imagine if you were to apply all style elements using inline styles. If you then decided to make even a simple change, you'd have to search through the entire application code to find the inline styles that need to be changed. Then, you'd have to recompile the program.

In contrast, external style sheets make it easy to change the appearance of your GUI. All you have to do is edit the `.css` file, and the formatting automatically reflects your changes.

Creating a Style Sheet

Now that you know how to attach styles to an application, scene, or individual node, it's time to turn your attention to the task of actually creating styles. As I mention earlier, a *style sheet* is a simple text file with the extension `.css`. You can create your style sheets with any standard text editor, including full-featured development studios, such as Eclipse or NetBeans, as well as simple text editors, such as TextPad or Notepad. Save the `.css` file in the same folder as the application's `.java` folder.

A style sheet consists of one or more *style rules* that determine the formatting that's applied to various types of elements in a scene. Each style rule consists of a *selector,* which determines which elements the style rule applies to, followed by a *declaration block,* which is a list of *style declarations* contained within a pair of braces. Each declaration consists of a property *name* followed by a colon and a value. Each declaration is terminated by a semi-colon.

For example:

```
.root
{
    -fx-background-color: lightgray;
    -fx-font-family: "serif";
    -fx-font-size: 12pt;
}
```

Here, the first line (`.root`) indicates that the style applies to all nodes in the scene. The declaration block includes three declarations, which supply values for the three properties named `-fx-background-color`, `-fx-font-family`, and `-fx-font-size`.

The following sections provide additional details about selectors and declarations.

Using type selectors

The most commonly used variety of selectors is a *type selector;* it corresponds to a JavaFX node type, such as `Button` or `TextField`. Type selectors begin with a period followed by the *style class name,* which is associated with all JavaFX node types. (***Note:*** The terms *style class* and *style type* are used interchangeably.)

For most controls, the name of the style class is similar to the name of the corresponding JavaFX class. To convert a JavaFX class name to a CSS style class name, use all lowercase letters and use a hyphen between words if the JavaFX class name consists of two or more words. The following list includes the CSS style class name for most of the JavaFX classes that have been presented so far in this book.

JavaFX Class	*CSS Style Class*
`Button`	`button`
`CheckBox`	`check-box`
`ChoiceBox`	`choice-box`
`ComboBox`	`combo-box`
`Label`	`label`
`ListCell`	`list-cell`
`ListView`	`list-view`
`Menu`	`menu`
`MenuBar`	`menu-bar`
`MenuButton`	`menu-button`
`MenuItem`	`menu-item`
`RadioButton`	`radio-button`
`Separator`	`separator`
`TableView`	`table-view`
`TextField`	`text-field`
`ToggleButton`	`toggle-button`
`Tooltip`	`tooltip`
`TreeCell`	`tree-cell`
`TreeView`	`tree-view`

Creating your own style class names

Every node has a `getStyleClass` method that returns an observable list of style class names. As a result, a given node can have more than one style class name. This can come in handy for scenes that have complicated formatting requirements because it allows you to group controls together for formatting purposes. For example, you can create additional class names to use for buttons if you want one set of buttons to be formatted differently than another set of buttons.

For example, suppose you want to set the font size for some buttons to 16 points. You could do that by creating a style type called `button-large` in the style sheet, like this:

```
.button-large
{
    -fx-font-size: 16pt;
}
```

Then, you could add the `button-large` style class to the list of style classes for the buttons you want formatted with larger type. For example:

```
Button btn1 = new Button("Wow!");
btn1.getStyleClass().add("button-large");
```

When a node has more than one style class name, all the class names will be used when matching selectors in the style sheet. In other words, the buttons that have the additional class named `button-large` will match style rules for both `button` and `button-large`.

Note: For many JavaFX node classes, the default style class collection is empty. For example, layout panes, such as `HBox` and `BorderPane`, do not have a default style class, nor do shape classes such as `Rectangle` or `Circle`. If you want to apply a CSS style to one of these nodes using style types, you must call `getStyleClass().add` to create a style class name for the node.

Using id selectors

If you want to create a style that applies to one and only one node in your scene graph, you can give that node a unique id by calling the node's `setId` method, like this:

```
Button btnOK = new Button("OK");
btnOK.setId("btn-wow");
```

Then, you can create a style rule that applies only to the node whose id is `btnOK`. In the selector, you must prefix the id with a hash mark, like this:

```
#btn-wow
{
    -fx-font-weight: bold;
}
```

The hash mark (#) is not used in the `setId` method to create the id, but it is required in the style sheet.

Node ids must be unique across the entire scene graph. In other words, you cannot create two nodes with the same id. Unfortunately, JavaFX does not enforce this, so it's up to you to make sure that your node ids are unique.

Using multiple selectors

A *style selector* can list more than one style type or id. To do that, you list all the types or ids as part of the selector, separating them with commas. For example, here is a style that's applied to all buttons, radio buttons, and check boxes:

```
.button, .radio-button, .check-box
{
    -fx-font-family: "serif";
}
```

Here's an example that includes several ids:

```
#btn1, #btn2, #btn3, #btn4
{
    -fx-fill: GREEN;
}
```

Specifying Style Properties

Within the declaration block of a style rule, each declaration specifies a style property and a value. For example, to set the font size to 12 points, use `-fx-font-size` as the property name and `12pt` as the value. In all, hundreds of style properties exist. Not all properties apply to all node types, however. Thus, each JavaFX node class has its own set of style properties.

All JavaFX style properties begin with the prefix `-fx-`. The following sections describe some of the more commonly used style properties.

Specifying font properties

For nodes that display text, you can use the properties shown in Table 12-1 to control the text style.

Table 12-1 Font Style Properties

Property	*Value*
`-fx-font-family`	The actual name of the font, or one of the following generic font types: `serif`, `sans-serif`, `cursive`, `fantasy`, or `monospace`.
`-fx-font-size`	A number followed by the unit of measure, which is usually `pt` (points) or `px` (pixels).
`-fx-font-style`	`normal`, `italic`, or `oblique`.
`-fx-font-weight`	`normal`, `bold`, `bolder`, `lighter`, `100`, `200`, `300`, `400`, `500`, `600`, `700`, `800`, or `900`.
`-fx-font`	A shorthand property that combines all other properties mentioned here into a single value that lists the style, weight, size, and family. Separate the values with spaces. If you want, you can omit the style and weight.

The following example sets the font for all button controls:

```
.button
{
    -fx-font-family: sans-serif;
    -fx-font-size: 10pt;
    -fx-font-style: normal;
    -fx-font-weight: normal
}
```

This version does the same thing using the shorthand `font` property:

```
.button
{
    -fx-font: 10pt sans-serif;
}
```

Specifying background colors

The `Region` class has a property named `-fx-background-color` that lets you specify the background color. Because both the `Layout` and `Control` classes inherit `Region`, you can use this property with any layout pane or control.

To apply a background color to a layout pane, you must first give the layout pane a style class name or id so that you can refer to it in a selector.

The following paragraphs describe the possible values you can supply for this property:

- **Named color:** JavaFX defines 148 distinct colors by name, including basic colors such as `black`, `white`, `red`, `orange`, and `blue` as well as exotic colors such as `cornsilk` and `thistle`.

  For example:

  ```
  -fx-background-color: red
  ```

  or

  ```
  -fx-background-color: papayawhip
  ```

  For a complete list of all 148 named colors, consult the CSS reference page online at `http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html#typecolor`.

- **RGB color:** The red-green-blue number of the color. This is usually expressed in hex, with two hex digits for each component of the color. The entire thing is prefixed with a hash like this:

  ```
  -fx-background-color: #f5f5f5
  ```

- **Gradient:** Lets you specify the color as a gradient that creates a smooth transition from one color to another.

  For information about creating gradients, flip to Chapter 13.

- **Lookup color:** A *lookup color* lets you define a set of color names in the `.root` section of the style sheet and then refer to the color name anywhere else within the style sheet.

You can create any name you wish for the color, provided the name doesn't conflict with a JavaFX property. For example:

```
.root
{
    my-color: aliceblue;
}

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

Here, `aliceblue` will be used as the background color for all buttons.

Specifying border properties

The `Region` class also has several style properties that let you create a border around the region. These properties allow you to add borders to layout panes or to add or change the borders in controls. Table 12-2 lists the border style properties.

Table 12-2 Border Style Properties

Property	*Value*
`-fx-border-width`	A number followed by the unit of measure, usually expressed in pixels (`px`)
`-fx-border-style`	`none`, `solid`, `dotted`, or `dashed`
`-fx-border-color`	A color

For example, the following style rule applies a dashed border to a style class named `bordered`:

```
.bordered
{
    -fx-border-width: 4px;
    -fx-border-color: black;
    -fx-border-style: dashed;
}
```