Juan Sanchez
Andy McIntosh

Foreword by
Narciso Jaramillo, Product Designer, Adobe Flex Team

# Creating Visual Experiences with Flex 3.0

Developer's Library

*Library of Congress Cataloging-in-Publication Data*

Sanchez, Juan, 1980-
Creating visual experiences with Flex 3.0 / Juan Sanchez, Andy McIntosh. – 1st ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-54537-0 (paper back : alk. paper)
1. Flex (Computer file) 2. Internet programming. 3. Application software—Development.
4. Web site development—Computer programs. I. McIntosh, Andy, 1982- II. Title.

QA76.625.S23 2008
006.7'6—dc22

2008033095



ISBN-13: 978-0-321-54537-4
ISBN-10: 0-321-54537-0

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, November 2008

**Editor-in-Chief**
Karen Gettman

**Senior Acquisitions Editor**
Chuck Toporek

**Development Editor**
Sheri Cain

**Managing Editor**
John Fuller

**Project Editor**
Sally Gregg

**Copy Editor**
Kelli M. Brooks

**Indexer**
Jack Lewis

**Proofreader**
Diane Freed

**Technical Reviewers**
Gary Mangum
R.J. Owen

**Interior Designer**
Gary Adair

**Cover Designer**
Gary Adair

**Composition**
Kim Arney

# Foreword

Over the past few years, the boundaries of what designers and developers can create on the Web have expanded enormously. Web sites increasingly integrate pixel-perfect visual design, video, animation, and advanced user interaction to create rich experiences that are both fun and useful. Many of these sites are built using the Adobe Flash platform. And more and more Flash sites are being built using Adobe Flex.

One of the most common misconceptions about Flex is that it's different from Flash. It's not: Flex is Flash. Flex applications run in the Flash Player; Flex is built on top of the Flash file format, language, and API. Through Flex, you have access to all the richness of the Flash Player, with the added benefit of being able to work more seamlessly with traditional development tools.

If you looked at Flex applications a couple of years ago, you might not have believed this seamless story. Flex applications looked like, well, Flex applications. They looked pretty nice right out of the box, but once you'd seen a few, you could instantly spot others.

Why was this? The focus of the first version of Flex was to make the Flash platform accessible to traditional software developers, and to make it easy for them to produce applications that looked good right out of the box. And it worked: People who would never have even conceived of using Flash began building enterprise-quality applications using our technology. Flex removed their barrier for entry to developing rich applications.

But you can only do so much in a release, especially a 1.0 release, and we didn't have time to also reinvent the world of *designing* rich applications. We did a lot to make the built-in component skins flexible—providing a large set of style parameters to allow people to tweak the visual appearance of components—but we knew that in order to really bring designers into Flex, we would need to do a lot more.

So, in Flex 2 and 3, we've been making it easier to build great visual and motion design into Flex applications. In Flex 2, we created view states and transitions to help designers and developers organize the appearance and behavior of complex dynamic applications. We also added a constraint-based layout mechanism that makes it easy to go from a pixel-perfect static design to a resizable application. In Flex 3, following the acquisition of Macromedia by Adobe, we built a streamlined skinning workflow between the Adobe Creative Suite tools and Flex Builder, and added a CSS design view to Flex Builder that lets you visually tweak the built-in appearance of components.

Andy McIntosh and Juan Sanchez are the perfect guides for your exploration into the visual design features of Flex. As designers who have crossed over into the world of development, and with their years of experience at EffectiveUI and other rich application design and development studios, they have been deeply engrossed in the world of Flex. In this book, they start with the basics of what designers and developers need to know about getting Flex applications to look good, then dive into more advanced topics and realistic examples showing how to create the look and feel of a complex application.

If you're a designer who's willing to leap into writing code once in awhile, or a developer with an eye for design, this book will help you build great Flex interfaces yourself. If you're a designer who never touches code, but wants to understand what your Flex

developers will be able to deliver, or if you're a pure developer who needs to learn how to implement the beautiful mockup a designer gave you, this book can help you work more effectively with your design/development partner.

Of course, Adobe isn't stopping here. We've got a lot of great stuff coming in Flex 4 and beyond to make it even easier for designers and developers to work together to build even richer experiences. But with this book in hand, you can get started today designing and building engaging applications. Go forth and create!

*Narciso (nj) Jaramillo*
*Product designer, Adobe Flex Team*

# Preface

The Web is always evolving as new technologies and greater user demands bubble to the surface. With the introduction of Flash in 1996, Web pages started to change from a static point-click-reload experience to a dynamic experience void of page refreshes and back button reactions. Since then, many technologies have risen to meet the ever-growing needs of the user, each proposing new conventions for reaching a final goal. As Flash matured, it became more frequently used for replicating interactions a user might expect from a robust desktop application. Flash was a usable solution for creating browser-based applications, but not necessarily the most approachable.

Now with Adobe Flex, the Flash player has matured to the point that it can be considered a target for enterprise-level applications. Along with Flex, there have been lots of competing technologies breaking onto the scene, underscoring the importance of rich user experience. New focus has been placed on the processes surrounding innovation, collaboration between designer and developer, usability, and deployment. These advancing facets pose new challenges for both designers and developers to overcome in order to create compelling visual experiences in Flex 3.

*Creating Visual Experiences with Flex 3.0* is a compilation of knowledge gathered from a myriad of real-world scenarios involving customization of Flex applications and creating rich user experiences.

## Scope of This Book

Rarely is there a case when no visual customization is required while creating a Flex application. It can be hard to find all the necessary information that covers the many approaches that can be taken to customize the visual experience of a Flex application. Interpreting that information to an actual use-case within an application can also be a daunting task. Our focus is to expose the knowledge we wished we had when we first started working with Flex. It is by no means meant to be an end-all be-all, but a starting point for your journey into working with the visual aspects of Flex 3. The goal is to give enough information to answer immediate questions and directions to find additional answers.

Everything discussed in this book can be used in some way, shape, or form, from robust enterprise-level applications to a simple RSS reader. The walkthroughs in this book are meant to be to-the-point and clearly communicate the approaches discussed. Since we learned Flex by pulling bits and pieces from a number of different examples to meet the needs of our applications, there is an *a la carte* presentation of walkthroughs. This will make it easy for you to grab what you need without foraging through massive lines of code to find the one bit you're interested in.

Most of the topics discussed in this book surround features immediately available in Flex. Creating custom components, advanced data visualization, and nonvisual components will not be covered. Although those things are also essential to enhancing the user experience, it extends beyond the scope of this book.

This book is not meant to be a rule-book of standards or a rigid set of guidelines. Each project is different and has its own requirements. There is always a judgment call to

be made between deadlines, budget, client needs, and user feedback that will dictate what approach you may take to implementing your own version of a unique visual experience. In the end though, you want to be happy with what you release, from the back-end architecture to the customer facing offering.

## Audience for This Book

The primary audience is designers and developers interested in translating design visions to Flex while maintaining the highest fidelity. Topics discussed revolve around some light design theory of rich user experiences and the visual presentation of user interface components. Designers and developers can use the information in this book to gain an understanding of the advanced level of customization that can be implemented in Flex. Also, those who may not be fully immersed in Flex development may find this book helpful in understanding what can be achieved visually with Flex.

The level of knowledge required for this book ranges from beginner to somewhat advanced, but the approach is always to make sure there is a guided sense of understanding. In the process of writing this book, we found ourselves referring to our own draft versions of chapters to find answers to a variety of questions that arose while working on a client project in Flex. We can only hope that this book may also serve as a solid reference that can help guide you in your own projects.

## Background

As Flex developers who are pretty involved with the Flex community, we get questions all the time through email and blog comments about the nuances of customizing Flex applications. It is a rare occurrence to *not* get approached at least once at a conference by someone with a laptop eager to find an answer or to gain some insight. In fact, this whole project was started based on a co-presentation that was given at a Flex-focused conference, called 360|Flex. The thought was to bottle our collective knowledge in a way that was more approachable than reading documentation and span the concepts we valued as we worked to implement our designs in Flex.

## How This Book Is Organized

The beginning of this book sets the stage for the rest of the chapters by introducing the capabilities of Flex and discussing some of the design foundation behind creating a visual experience. The remaining chapters go into more technical detail about visual customization and implementation in Flex. Because each chapter discusses techniques that may complement others, chapters have been ordered to take advantage of material presented in previous chapters. This is not to say that you can't jump into any given chapter at any point throughout your reading.

**Chapter 1: Rich Internet Applications.** Design plays a large role in the final appearance of your Flex application. This chapter gives a general overview of design fundamentals and thinking points that can transfer to your application's final design.

**Chapter 2: Adobe Flex and AIR.** An overview of Adobe Flex 3 and the Adobe Integrated Runtime (AIR) sets the stage for some of the technical aspects in the remainder of the book.

**Chapter 3: Dynamic Layout.** Layout defines the visual structure of your application. Learn about the various components and techniques you can use to create the visual skeleton of your application.

**Chapter 4: Styling.** Using styling properties, you can customize components rapidly and easily. Styling lays the groundwork for other methods when customizing a user interface.

**Chapter 5: Graphical Skinning.** When styling a component isn't enough, skinning can step in and take customization to the next level. You can leverage existing graphics applications like Adobe Illustrator, Flash, Fireworks, and Photoshop to customize the look of your Flex application.

**Chapter 6: Programmatic Skinning.** Drawing graphics programmatically to be used for the appearance of components is very powerful, but also more complex than graphical skinning. Learn how you can leverage the power of programmatic skinning to create customizable interfaces.

**Chapter 7: Lists and Navigation Components.** The additional challenges for customization posed by List and Navigation controls, including item renderers and a data-driven display, are addressed. Lists and navigation components present different challenges for customization because they are primarily data driven and comprise many pieces.

**Chapter 8: Indicators and Cursors.** Guiding a user through an application is instrumental to a user achieving a goal. Steer your customers in the right direction by adding another level of customization.

**Chapter 9: Fonts and Text.** The way you display text and how it looks can play an emotional role as well as an informational one. Establishing structured and well-placed text is well within reach when working with Flex.

**Chapter 10: Filters and Blends.** Adding a sense of layering and depth can add a level of richness to an application. Flex provides a number of filters and blends that can be applied to components for a variety of purposes.

**Chapter 11: Effects and Transitions.** By deploying in the Flash player, Flex applications can capitalize on the advantages of using motion to create a fluid and immersive experience. Learn how you can leverage motion-based features of Flex in a variety of ways.

**Chapter 12: Flex and Flash Integration.** Flash and Flex share a common ground: ActionScript 3 and the Flash player. Flash CS3 and plugins can be used to create custom components, skins, containers, and motions.

**Chapter 13: Customizing AIR Applications.** AIR allows you deploy desktop applications using the same code-base as Flex applications. However, AIR applications have additional parameters that can be customized and are specific to the desktop environment.

## Conventions Used in This Book

As you read this book there are a number of conventions used to guide you through varying types of information. You will find the code hosted at http://www.cveflex.com. At any given point you may encounter the following:

**Tables.** Used to group items and descriptions in a structured format for quick and easy reference.

**Figures.** Range from screenshots to diagrams to graphical instructions.

**Listings.** Code that supports discussed concepts and can include MXML, CSS, ActionScript, and comments.

**Exercises.** Longer walkthroughs that may involve code, graphic creation, traversing multiple applications, and running sample applications.

**Tips.** Used as pointers for clarifications and suggestions.

**Notes.** Used to provide further information.

In every case, these items are noted by their type, chapter number, and a sequential number (e.g., Table 5-3, Figure 7-13, etc.). For Exercises, a number for the exercise is introduced (e.g., Figure 5.1-4, Listing 4.3-7). This convention is used to refer to these items in the text and to allow you to cross-reference things as you move through the book.

## Styling and Skinning Diagrams

You will frequently be referred to Appendix A's, "Skinning and Styling Diagrams," throughout the book. This reference is a series of diagrams that point out the customizable parts of visual Flex components, including containers and controls. It is meant to be a complementary reference during the process of customizing components in your Flex application.

## Additional Resources

Other references you will find include Appendix B, "Filters Cheat Sheet," and Appendix C, "Resources and Cool Stuff." These cheat sheets are meant to act as quick references that expose the properties for these Flex 3 features along with a brief description of the property.

# 4

# Styling

Every visual component in the Flex framework has a default appearance, including colors, shape, font, and more. Although this default look, called the Halo theme, looks nice as a default, a custom look can help your application stand out among other Flex applications. The Flex framework allows you to customize the look of components using special properties called styles. Styles can define things like colors, fonts, and other visual characteristics, which makes styling one of the most powerful mechanisms in Flex, but also one of the most complex mechanisms. Mastering styling is key to creating expressive Flex and AIR applications.

Using styling you can easily change the appearance of your application. Each Flex component has its own set of style properties that allow you to alter different parts of it to create a custom look. If you're looking for the quickest way to divert from the default Halo Flex theme without too much effort, styling is the way to get there. Figure 4-1 shows variations of a button made to look different using just style properties, starting with the default look.


Figure 4-1 Visual variations of a button changed using styling

## Understanding Style Properties

Not all properties of visual components can be specified as styles. Typically, style properties are those that modify only the appearance of a visual component such as color-related properties, font and font size, padding, and so on. Table 4-1 shows the types of properties that can be classified as styles and those that cannot. Style properties are used to specify skins and other visual elements (see Chapter 5, "Graphical Skinning," for more about skinning).

**Table 4-1 Flex Properties That Can Be Applied as Styles and Those That Cannot**

| | | |
|---|---|---|
| **With styling you can specify:** | | |
| Fonts | Text formatting | Colors |
| Icons, indicators and cursors | Skins | Alignment |
| Relative positioning | Padding | |
| **With styling you cannot specify:** | | |
| Absolute positioning | Size | Event handlers |
| Effects and transitions | States | Filters |
| Component properties | | |

## Inheritance

Components may inherit styles from their parent containers. Consider a Button component within a Canvas container where the Canvas has the `color` style set to `0xFF0000` (red) as shown in Listing 4-1. Even though `color` has not been set directly on the Button component, it will still have a red label, just by virtue of being a child of the Canvas (see Figure 4-2).

Listing 4-1 **The value of the color style that is set on the Canvas component will cascade to the Button component within**

```
<mx:Canvas
      width="400"
      height="100"
      backgroundColor="0xFFFFFF"
      color="0xFF0000"
      >
      <mx:Button
            label="Button"
            verticalCenter="0"
            horizontalCenter="0"
            />
</mx:Canvas>
```


Figure 4-2 Button inheriting the color property from a canvas

Inheritance is a very powerful function of the styling mechanism, as it allows top-level style changes to propagate throughout the application. This behavior, coupled with the various CSS selectors discussed in the upcoming "Applying Styles with CSS" section, make it possible to change the visuals of an entire application rather easily.

> **Note**
>
> Not all style properties are inheritable. Refer to the Adobe Flex 3 Language Reference for more information.

## Data Types and Formats

With the exception of class references and embedded assets, styles can only be of type String, Number, or an Array of Strings and Numbers. To validate values, styles have a format associated with them. For example, the `backgroundColor` style of a Canvas is associated with the "color" format, which means it can be specified as either hexadecimal, RGB, or qualified color name.

Table 4-2 describes the relationship between various data types, formats, and the units/syntax available when specifying styles.

> **Note**
>
> Embedding assets and using class references are discussed at length in Chapter 5 and Chapter 9.

Table 4-2 **Property Formats**

| Name | Sample styles | Units / Syntax |
|---|---|---|
| **Length (Number)**<br>**style-name: length [unit]** | `fontSize`<br>`borderThickness`<br>`horizontalGap`<br>`paddingTop`<br>`paddingBottom`<br>`paddingLeft`<br>`paddingRight` | pixels (px)<br>inches (in)<br>centimeters (cm)<br>millimeters (mm)<br>points (pt)<br>picas (pc)<br>keywords, for fontSize only:<br>• xx-small<br>• x-small<br>• small<br>• medium<br>• large<br>• x-large<br>• xx-large |
| **Time (Number)**<br>**style-name: time** | `openDuration`<br>`closeDuration`<br>`selectionDuration` | milliseconds |

Table 4-2 **Continued**

| Name | Sample styles | Units / Syntax |
|---|---|---|
| **Color (String or Number)**<br>**style-name: color** | `backgroundColor`<br>`borderColor`<br>`color`<br>`textSelectedColor` | hexadecimal (#000000 - #FFFFF)<br>RGB (r%, g%, b%)<br>VGA color names:<br>• Aqua<br>• Black<br>• Blue<br>• Fuschia<br>• Gray<br>• Green<br>• Lime<br>• Maroon<br>• Navy<br>• Olive<br>• Purple<br>• Red<br>• Silver<br>• Teal<br>• White<br>• Yellow |

# Applying Styles Inline

The simplest way to apply a style is to set a style directly on an instance of a visual component. Using MXML, the syntax is the exact same as setting a property (see Listing 4-2).

Listing 4-2 **Specifying the color style on a Button component inline with MXML**

```
<mx:Button color="0xFF0000" label="Red" />
```

> **Note**
>
> Note that syntax for setting the `color` style is the same used for setting the `label` property, which is not a style.

Style properties can be bound to variables, making it possible to create interactive or dynamic interfaces. For example, the `fontSize` property of a TextArea component could be bound to the value property of a NumericStepper. Interacting with the NumericStepper would affect the size of the text in the TextArea.

# Applying Styles Using CSS

Although applying styles inline is easy, the real power of the styling framework is that you can define sets of styles, called selectors, and apply them to many component instances. Selectors are defined using CSS (cascading style sheets) syntax written in either external style sheets or in between `<mx:Style>` tags within a component.

The CSS syntax used in Flex is similar to that used in HTML; however, there are some important differences you should be aware of. This section discusses local and external styles, the various available selectors, and compares Flex CSS to HTML CSS.

> **Note**
>
> When working with styles using CSS, you have the option to reference styles using either inter-cap notation (someStyle) or hyphenated notation (some-style). Flex Builder's code hinting for styles automatically completes style names using hyphen notation.

## Understanding Local and External Styles

Styles can be defined using CSS in either an external style sheet or within MXML `<mx:Style>` tags. Styles defined in the latter mode are available only to the components and their children that are created within that particular MXML file. Using an external style sheet is recommended because it keeps all the styles centralized, which can increase legibility and simplify maintenance. Also, external style sheets can be packaged into themes and then dynamically loaded to change the look of an application without needing to recompile the application, as described later in this chapter.

Unless otherwise specified, a Flex application loads the standard Halo theme. The style sheet associated with this theme is located at [Adobe Flex Builder 3]/sdks/3.0.0/frameworks/projects/framework/defaults.css. Although this style sheet doesn't define every property for all components, it does define a lot of them. If you replace this file or load a different style sheet, your application may look pretty plain unless you redefine a lot of styles.

> **Tip**
>
> Studying the defaults.css style sheet is a great way to become familiar with the elements of Flex applications that can be stylized.

## Understanding CSS Selectors

Style definitions, whether created externally or locally, comprise a set of styles, called selectors. Based on how they're named, they are applied to components in one of several ways: per instance (class selector), per class (type selector), or globally (global selector). Listing 4-3 provides the general syntax of how styles are defined in CSS.

Listing 4-3 **Typical CSS selector syntax**

```
style_name
{
      style_property: value;
      [...]
}
```

### Class Selectors

Class selectors define a set of properties that can be applied to specific instances of components. For example, a class selector named `bigRed` may specify `fontSize` of 24 and color value of `0xFF0000` (see Listing 4-4). These properties can then be applied to any instance of a component by setting the `styleName` property to `bigRed` (see Listing 4-5).

Listing 4-4 **The bigRed class selector definition**

```
.bigRed
{
      fontSize: 24;
      color: #FF0000;
}
```

Listing 4-5 **The bigRed class selector being applied to an instance of button and an instance of ComboBox**

```
<mx:Button styleName="bigRed" />
<mx:ComboBox styleName="bigRed" />
```

> **Note**
>
> The name of class selectors must start with a period (.) when they are defined, but the period is omitted when applying the styles. Class selector names typically use inter-caps, with a lowercase first letter.

### Type Selectors

Type selectors define a set of properties that are automatically applied to all instances of a certain component. For example, a Button type selector that specifies a color value of `0xFF0000` causes all instances of button to have red text (see Listing 4-6).

Listing 4-6 **A Button type selector specifying red text**

```
Button
{
      color: #FFFFFF;
}
```

> **Note**
>
> The name of type selectors must exactly match the name of the MXML file or ActionScript class that defines the component; therefore, they typically begin with an uppercase letter, use inter-caps, and are not preceded by a period (.).

### The Global Selector

The global selector is a special type selector that can impact nearly every component of a Flex or AIR application. Basically, if no other styles have been defined for a component, a value from the global selector will be applied, if available.

Because Application (or WindowedApplication in AIR) is typically the topmost container, defining a type selector for Application seems to behave very similarly to using the global selector. However, the key difference is that setting a style property in the global selector can affect even noninheriting properties. For example, setting `cornerRadius` to 8 in the global selector affects all components that have a `cornerRadius` style property, even though `cornerRadius` is not an inheritable style (see Listing 4-7). Setting `cornerRadius` to 8 in only the Application selector will not impact any components.

Also, the global selector impacts components that are not actually children of the main Application container, such as popup alerts or windows.

Listing 4-7 **The special global selector specifying a `cornerRadius` of 8 pixels**

```
global
{
      cornerRadius: 8px;
}
```

> **Note**
>
> The global selector does not need to be prefaced with a period (.) because it is a type selector, but it must be all lowercase.

## HTML versus Flex CSS

If you've used CSS in HTML, the syntax for CSS in Flex will be familiar, but you'll find that the capabilities differ. Flex does not actually allow you to "cascade" styles in CSS as you can with HTML. For example, the CSS code snippet in Listing 4-8 will *not* apply styling to a button inside a component called SearchInput. CSS in Flex also does *not* support pseudo selectors, like `img:hover`. This means the code in Listing 4-9 would not affect the over state of a button in Flex, as it might in HTML. Furthermore, you cannot specify dimensions (such as `width` and `height`) and positioning (such as `x` and `y`), as you can in HTML.

Listing 4-8 **Cascading styles, as depicted here, will not work in the Flex CSS implementation**

```
SearchInput Button
{
      color: #FFFFFF;
      fillColors: #72CFFF, #165D81;
}
```

Listing 4-9 **Pseudo selectors, as shown here, will not work in the Flex CSS implementation**

```
Button:over
{
      color: #222222;
      fillColors: #72CFFF, #165D81;
}
```

# Style Precedence

Depending on where and how they are defined, some styles may override others. Leveraging this gives you a lot of control over how styles are applied in your application, while still allowing you to make broad-stroke changes at a high level.

Redefining a selector does not completely replace the selector, but rather appends additional properties and overrides any existing ones. This means you can simply override the properties you wish, and keep everything else as it's been defined previously. For example, you like the way the Halo buttons look, but you want to shrink the corner radius to 0 pixels. All you have to do is create a Button type selector and set the `cornerRadius` property to 0 (see Listing 4-10). All instances of Button maintain the properties defined in the default style sheet, but will not have rounded corners (see Figure 4-3).

Listing 4-10 **Overriding the cornerRadius value for all Button components**

```
<mx:Style>

      Button
      {
            cornerRadius: 0px;
      }

</mx:Style>

<mx:Button label="Button 1"/>
<mx:Button label="Button 2"/>
```


Figure 4-3 Halo buttons without rounded corners

Likewise, assigning a class selector to a component for which a type selector is also defined results in a combination of the style properties being applied. Listing 4-11 extends the previous example by creating a type selector and applying it to the second button. Both buttons still have square corners, but the second one also has a red label as specified by the type selector (see Figure 4-4).

Listing 4-11 **Overriding the cornerRadius value for all Button components and applying a class selector to a specific button**

```
<mx:Style>

      Button
      {
            cornerRadius: 0px;
      }

      .redText
      {
            color: #FF0000;
      }

</mx:Style>

<mx:Button label="Button 1"/>
<mx:Button label="Button 2" styleName="redText"/>
```


Figure 4-4 Halo buttons without rounded corners. Button 2 has a red label.

Generally speaking, styles applied inline override styles defined locally, which override styles defined in external style sheets. Moreover, inline styles override class selectors, which override type selectors, which override the global selector. Whew. Let's consider some examples.

Let's say you've defined that all buttons in your application should have a dark gray label by specifying the color style property in an external style sheet named myStyles.css (see Listing 4-12). However, in a particular view, like a high-contrast heads-up display panel (HUDPanel.mxml), you want all the buttons to have white labels. One way to accomplish this is to redefine the `color` property for all buttons in an `<mx:Style>` block in the MXML file that defines the panel (see Listing 4-13).

Listing 4-12 **Button label color specified in an external style sheet**

```
/* myStyles.css */

Button
{
      color: #333333 /* dark gray */
}
```

Listing 4-13 **Overriding the Button type selector within a specific view**

```
<!– HUDPanel.mxml –>

<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml" layout="">

      <mx:Style>

            Button
            {
                  color: #FFFFFF; /* white */
            }

      </mx:Style>

</mx:Panel>
```

You could also approach the challenge described previously by creatively using the different types of selectors. A Button type selector could still define the dark grey label, but you could create a class selector called hudButton (see Listing 4-14) and apply that to each button within the heads-up display panel (see Listing 4-15).

Listing 4-14 **Defining a type selector and a class selector**

```
/* myStyles.css */

Button
{
      color: #333333 /* dark gray */
}

.hudButton
{
      color: #FFFFFF /* white */
}
```

Listing 4-15 **Overriding the Button label color by applying the class selector**

```
<!— HUDPanel.mxml —>

<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml" >

      <mx:Button styleName="hudButton" label="click me" />

</mx:Panel>
```

Yet another way to approach this challenge is to utilize inheritance. Remember that the color style property propagates from a parent container to the children within. So, perhaps the easiest way to solve this problem is to set the `color` property for the entire HUDPanel container with a type selector and let the buttons inherit the value (see Listing 4-16). Keep in mind that this will change the `color` property of all eligible children components such as Labels and ComboBoxes, but this is probably desirable.

Listing 4-16 **Specifying the color for all components in the HUDPanel container with a type selector**

```
/* myStyles.css */

Button
{
      color: #333333 /* dark grey */
}

HUDPanel
{
      color: #FFFFFF /* white */
}
```

> **Tip**
>
> If you have similar styling for various style selectors, you can combine them using comma delimitation to cut down on the amount of code you have to write as well as maintain consistency. The CSS code in Listing 4-17 shows similar style properties for .downloadButton and .saveButton being grouped together, but redefined separately for their differences, in this case, icons.

**Listing 4-17 Combining styles using comma delimitation**

```
.downloadButton, .saveButton
{
      fillColors: #FFFFFF, #CCCCCC;
      borderColor: #666666;
      cornerRadius: 8;
      color: #222222;
}

.downloadButton
{
      icon: Embed(source="downloadIcon.png");
}

.saveButton
{
      icon: Embed(source="saveIcon.png");
}
```

# Working with Styles Using ActionScript

So far, we've mostly discussed working with styles using MXML. If necessary, you can accomplish the same tasks, and more, using ActionScript. Perhaps your application requires a particular button to be red or green depending on the values of certain variables. You could execute a function that checks those variables and conditionally sets the `backgroundColor` style of your button. As with many things in application development, there are several ways to accomplish this, but let's have a look at how you might do it with ActionScript (see Listing 4-18).

To set a style on an instance of a component, you call the `setStyle` method available to any style-enabled component, passing it the name of the style property and the value you wish to set it to. This is basically the equivalent of specifying a style property inline in MXML. Because calling this method not only impacts the component on which it was called, but also any components contained within, it should be used judiciously. To get the current value of a style property using ActionScript, you use the `getStyle` method, which is a much lighter operation than `setStyle`.

Listing 4-18 **Conditionally setting style properties using ActionScript**

```
<mx:Script>
<![CDATA[
      public var valid:Boolean = true;
      public function changeButtonColor () : void
      {
            if ( valid == true )
            {
                  // make the background green
                  myButton.setStyle('backgroundColor',0x00CC00);
            }
            else
            {
                  // make the background red
                  myButton.setStyle('backgroundColor',0xCC0000);
            }
      }
]]>
</mx:Script>

<mx:Button id="myButton" click="changeButtonColor()" />
```

> **Warning**
>
> Because the `setStyle` method accepts the style name as a string, there is no validation as to whether that property actually exists. That means you could misspell the property name, and it would fail, and fail silently. You won't even get an exception when you try to set it.

## Creating Stylable Widgets

It is common to composite two or more components to create a reusable widget for your application. For example, you might extend an HBox and stuff it with a TextInput and Button to create a SearchInputWidget to use throughout your application. Listing 4-19 demonstrates exactly that to create a component that looks like Figure 4-5.

Listing 4-19 **Applying style to an element of a widget with an explicit class selector**

```
<!— SearchInput.mxml —>

<?xml version="1.0" encoding="utf-8"?>
<mx:HBox xmlns:mx="http://www.adobe.com/2006/mxml" >
      <mx:TextInput
            width="100%"
            height="100%"
            />
```

Listing 4-19 **Continued**

```
      <mx:Button
            label="Search"
            height="100%"
            styleName="searchInputButton"
            />
</mx:HBox>
```


Figure 4-5 A stylable SearchInput widget that combines a TextInput with a SearchButton

Notice in Listing 4-19 that the `styleName` of the button is set to `searchInputButton`. This approach exposes the style of the button so that it can be defined outside of the widget, but all instances of SearchInputWidget will have the exact same style. Although this is better than applying a bunch of inline styles to the button, there is still a better way. Changing the button definition as shown in Listing 4-20 allows you to customize each instance of the SearchInput widget differently (see Listing 4-21).

Listing 4-20 **Applying style to an element of a widget using a dynamic class selector**

```
...
<mx:Button
      label="Search"
      height="100%"
      styleName="{getStyle('buttonStyleName')}"
      />
...
```

Listing 4-21 **Applying two different styles to two instances of the same widget**

```
<local:SearchInputWidget buttonStyleName='redButton' />
<local:SearchInputWidget buttonStyleName='blueButton' />
```

For this trick to work, you must expose `buttonStyleName` as a style using metadata as shown in Listing 4-22. Style metadata tells the compiler which style can be used with a particular component, its format and data type, and whether children of this component should inherit this style.

Listing 4-22 **Style metadata used to define a buttonStyleName style**

```
[Style(name="buttonStyleName", type="String", inherit="no")]
```

### Introducing the Style Manager

Behind all styling operations in Flex is the StyleManager. It is possible to access, modify, and define CSS selectors using ActionScript and the StyleManager. This opens a world of possibilities for creating dynamic styles or performing other tricks.

You can use ActionScript to directly access the selectors from the StyleManager to take an inside-out approach to styling. For example, you might define a selector that embeds several icons (see Listing 4-23). Next, from within your application, use ActionScript to access the values of that selector (see Listing 4-24). It's a pretty handy trick because it keeps all of your icons in one place.

Without this trick, you have to embed your assets within the component in which it will be used, which can get ugly if that component is buried deep in the application package structure. Imagine a SearchInput widget located in a com.cve.view.controls package. A reference to an icon located in an images folder at the root of the project might look something like Listing 4-25.

The StyleManager also enables you to load and unload entire style sheets at runtime, which enables you to create dynamic, on-the-fly customizations of your Flex application. For this to work, your style sheets must be compiled as SWF. Runtime styling is described at length in Exercise 4.1.

Listing 4-23 **Several assets embedded in a single selector**

```
.icons
{
      wrenchIcon: Embed('images/wrench.png');
      searchIcon: Embed('images/magnifier.png');
      loginIcon: Embed('images/lock.png'):
}
```

Listing 4-24 **Accessing embed assets using the StyleManager**

```
<Button
    icon="{StyleManager.getStyleDeclaration('.icons').getStyle('wrenchIcon')}"
    label="Customize"
/>
```

Listing 4-25 **An ugly Embed statement within a component**

```
[Embed(source="../../../../images/magnifyer.png")]
public var searchIcon:Class;
```

## Styling in Design View

You don't always have to assign styling properties to components directly in the Source View of Flex Builder. In Design View, you can access all the style properties for any component in

the Flex Properties panel (Window > Flex Properties). With the Flex Properties panel visible, you can select any component to access visual selectors for the styles of that component. Figure 4-6 shows what the Flex Properties panel looks like when selecting a Button in Design View. You can also access other style properties in this same panel, as shown in Figure 4-7.

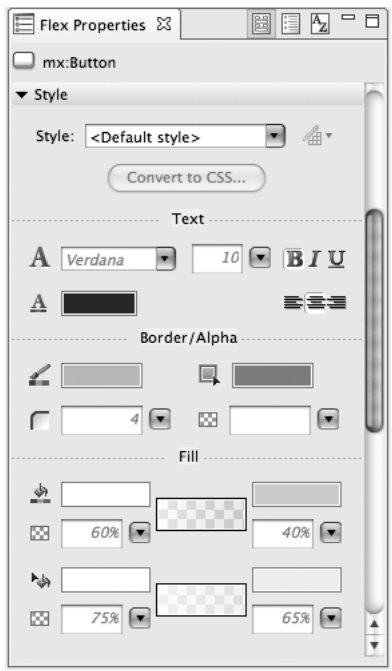


Figure 4-6 Styles for a button within the Flex Properties panel

**Note**

Not every style property is represented in the Style panel. To access more styles, click on the category list and scroll to the Styles category, or find a style via the alphabetized list of properties as shown in Figure 4-7.

Whenever you make changes to the styling of a component using the Flex Properties panel, the styles will be created inline of the component's MXML. However, you can easily convert your inline styling to CSS to use throughout your application. In Flex Builder, this is really easy to do by following these steps:

1. Once you have created styling for a particular component, select that component and, in the Flex Properties panel, click the Convert to CSS button as shown in Figure 4-8.


Figure 4-7 Accessing other style properties via the Flex Properties panel

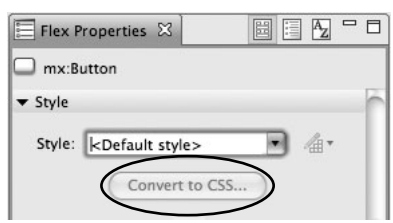


Figure 4-8 The Convert to CSS button

2. When you click the Convert to CSS button, the New Style Rule window appears (see Figure 4-9). If you don't have a CSS file already created, you can click the New button to create one to hold your new style.
3. With your CSS file specified, you can select to apply the styling globally, as a type selector, to a specific component, or to a component using a style name. You can refer to the "Understanding CSS Selectors" section in this chapter for more information on the differences between these options. After selecting your desired option, click OK.


Figure 4-9 The New Style Rule window

4. When you click OK, the appropriate CSS is written to the specified CSS file and you are taken into CSS Design View where you have additional options to customize your components using a more visual tool set.

**Tip**

You can access CSS Design View any time you have a CSS file open in Source View mode.

## CSS Design View

One of the new features of Flex Builder 3 is CSS Design View, which allows you to visually edit styles within a CSS file. In this view, you are presented with the various component parts that can be styled, as well as any available states. You can also do things like zoom in and out for pixel perfect accuracy, pan around using the Hand tool, or add and remove styles. Figure 4-10 is a snapshot of CSS Design View for the CSS of the styling of a button.

In CSS Design View, you can also easily assign other visual parameters like icons, fonts, and skins for your components. Refer to Chapters 5 or 9 for more information on using those features of CSS Design View.

In CSS Design View, you can also easily navigate between different styles. You can do this by selecting a style from the Style Combo Box. You can also move throughout the CSS code structure of your CSS file using the Outline panel (see Figure 4-11) when in CSS Source View, which you can access by selecting Window > Outline. Clicking on any of the properties in the Outline structure jumps you to the corresponding CSS code.

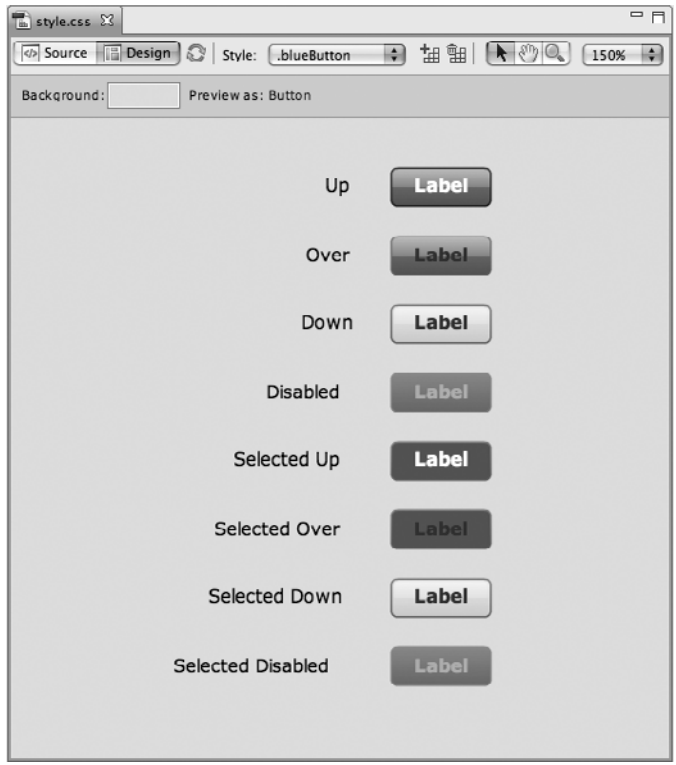


Figure 4-10 CSS Design View of a styled button

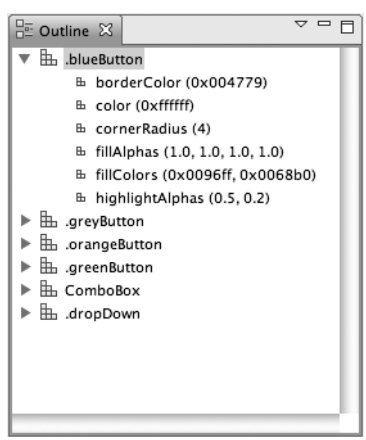


Figure 4-11 CSS Outline View with button styling and embedded font

Working in CSS Design View is a quick and easy way to get instant feedback on the look of a styled component. It's great for quick prototyping and rapid development. However, you may find it helpful to become familiar with working in the CSS Source View for code formatting and taking advantage of some of the tips discussed in this chapter.

# Themes

A theme is essentially a look, generally a visually consistent one, for a Flex application that has been created using CSS and/or supporting assets. A theme can vary in complexity from simple color changes to graphic-rich styling. After spending the time and effort to create a theme for your Flex application, you can package it to be repurposed or distributed by creating a theme.

At a minimum, a theme can be used by passing its assets to others interested in implementing it into their Flex application. You can also package those assets as a single theme within a Flash Component file (SWC) to be easily distributed or compile that theme SWC to be used as a theme loaded at runtime. However, once you create a theme SWC, it will no longer be editable.

A theme SWC is basically a package of any CSS files, graphic assets, or programmatic skin classes that comprise your Flex application's appearance. Each of the supporting graphical assets or classes must be either embedded or use a ClassReference in order to ensure their inclusion into the theme SWC. You can learn more about embedding graphic assets in the Skinning chapter. You can use the Flex compiler to apply a theme SWC and the compc utility to create one. To learn more about working with theme SWCs, refer to the Flex 3 documentation.

> **Note**
>
> As a reference for creating your own theme SWCs, you may want to duplicate the defaults.css file that comes with the Flex 3 SDK and build upon it to make your own custom theme.

# Summary

Styles are different from properties and exhibit several special behaviors, such as inheritance. The most powerful aspect of styling is that styles can be defined in sets and applied to components in a variety of ways.

Styling plays a major role in the way you define the look of your application. The more you work with it, the more you will see just how powerful it can be. Whether you just need to change the look of a single button or an entire application, styling is the root mechanism for complete customization.

The following chapters build on the concepts discussed here and introduce further customization techniques such as skinning and working with indicators, cursors, and text.

# Index

## Symbols and Numbers



## A

## B

## C

## D

## E

## F

## G

## H



## I

## J

## K



## L



## M

## N



## O

## P

## R

## S

## T

## U

## V



## W



## X



## Z