

Learning Flex™ 4

GETTING UP TO SPEED WITH RICH INTERNET APPLICATION
DESIGN AND DEVELOPMENT

Alaric Cole & Elijah Robison



Adobe
Developer
Library

O'REILLY®

In this book, we'll help you develop every skill you need to master Flex,
with fun and practical exercises—and instant results. —Alaric Cole

Learning Flex 4

Learn Adobe Flex 4 in a fun and engaging way with this book's unique, hands-on approach. Using clear examples and step-by-step coaching from two experts, you'll create four applications that demonstrate fundamental Flex programming concepts.

Throughout the course of this book, you'll learn how to enhance user interaction with ActionScript, and create and skin a user interface with Flex's UI components (MXML) and Adobe's new FXG graphics format. You'll also be trained to manage dynamic data, connect to a database using server-side script, and deploy applications to both the Web and the desktop.

Learning Flex 4 offers tips and tricks the authors have collected from years of real-world experience, and straightforward explanations of object-oriented programming concepts to help you understand how Flex 4 works.

Whether you're a beginner or an experienced developer coming to Flex from another platform, this book is the ideal way to learn how to:

- Work with Flash Builder 4 and the Eclipse IDE
- Learn the basics of ActionScript, MXML, and FXG
- Design a Flex application layout
- Build an engaging user interface
- Add interactivity with ActionScript
- Handle user input with rich forms
- Link Flex to a server with PHP and MySQL
- Gather and display data
- Style applications, and add effects, filters, and transitions
- Deploy applications to the Web, or to the desktop using Adobe AIR

About the authors

Alaric Cole is an early adopter of Flex, having worked with the Flash Platform since the introduction of ActionScript. As a consultant, he helps organizations build rich enterprise applications, interactive media, and advanced framework components.

Elijah Robison is a software engineer with VillaGIS, Inc., where he helps develop mapping applications using Adobe Flex and Adobe AIR. He offers a fresh perspective that helps to demystify Flex for beginners.

US \$44.99

CAN \$51.99

ISBN: 978-0-596-80563-0



5 4 4 9 9

Safari
Books Online

Free online edition
for 45 days with purchase of
this book. Details on last page.



Adobe
Developer
Library

oreilly.com

adobedeveloperlibrary.com

O'REILLY

Learning Flex 4

Getting Up to Speed with Rich Internet Application Design and Development

Alaric Cole and Elijah Robison

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Learning Flex 4

by Alaric Cole and Elijah Robison

Copyright © 2011 Alaric Cole and Elijah Robison. All rights reserved.

Printed in Quebec, Canada.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department:

800-998-9938 or corporate@oreilly.com.

Editor: Mary Treseler

Production Editor: Kristen Borg

Copyeditor: Genevieve d'Entremont

Technical Reviewers: Jodie O'Rourke and Russ Ferguson

Proofreader: Sada Preisch

Interior Designer: Ron Bilodeau

Cover Designer: Matthew Woodruff and Karen Montgomery, based on a series design by Mark Paglietti

Indexer: Lucie Haskins

Print History:

November 2010: First edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. This book's trade dress is a trademark of O'Reilly Media, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-80563-0

[TI]



Adobe Developer Library, a copublishing partnership between O'Reilly Media Inc. and Adobe Systems, Inc., is the authoritative resource for developers using Adobe technologies. These comprehensive resources offer learning solutions to help developers create cutting-edge interactive web applications that can reach virtually anyone on any platform.

With top-quality books and innovative online resources covering the latest tools for rich-Internet application development, the *Adobe Developer Library* delivers expert training, straight from the source. Topics include ActionScript, Adobe Flex®, Adobe Flash®, and Adobe Acrobat® software.

Get the latest news about books, online resources, and more at *adobedeveloper-library.com*.

CONTENTS

Preface	xi
---------------	----

Chapter 1

Getting Up to Speed	1
What Is Flex?	1
What About AIR?	4
Where Flex Fits	5
Why Use Flex?	6
How Flex Compares to Other Technologies	8
When Not to Use Flex	11
Summary	11

Chapter 2

Setting Up Your Environment.....	13
Using Alternatives to Flash Builder	14
Introducing Flash Builder and Eclipse	16
Running Your First Application	20
Creating a New Flex Project	27
Summary	29

Chapter 3

Using Design Mode	31
A Blank Slate: Your First Project	31
Adding Components to the Application	36
Exploring Common Components	39
Modifying Properties Directly	44
Summary	50

Chapter 4

Using Source Mode	51
What Design Mode Does	51
Anatomy of a Flex Application	52
Adding Components in Source Mode	54
Code Completion	55
MXML in Depth	56
S, FX, and MX: Namespaces Explained	60
Summary	62

Chapter 5

ActionScript Basics for Flex Applications	63
Getting Ready	63
Dot Notation	64
Inline ActionScript	65
Assignment and Concatenation	66
Functions	66
Variables	74
Data Types	75
Objects	78
Classes	79
ActionScript's Relationship with MXML	85
Comments?	87
Summary	88

Chapter 6

Debugging Flex Applications	89
Outputting Values to the Console Using trace()	90
Inspecting Event Properties with trace()	93
Using Breakpoints	95
Summary	101

Chapter 7

Adding Interaction with ActionScript	103
Understanding Events	104
Common Events	104
Researching Events	105
Listening for and Responding to Events	109
Collision! A Whirlwind of Events	120
Summary	136

Chapter 8

Using Data Binding	137
What Is Data Binding?	137
Applying Data Binding	138
Two-Way Bindings	146
Handling Complex Data with Data Models	147
When Data Binding Isn't Appropriate	150
Summary	150

Chapter 9

Designing Application Layouts	151
Types of Layouts	152
The Display List	155
Sizing	160
Controlling Whitespace in the Layout	162
Advanced Containers	165
Spacers and Lines	168
Alignment	171
Constraints-Based Layout	173
Summary	176

Chapter 10

Creating Rich Forms	177
Preparing a Form-Based Application	177
Validating Data	184
Restricting Input	196
Formatting Input	197
Combining Restrictions and Formatters	199
Linking Formatters to Functions	200
Summary	207

Chapter 11

Gathering and Displaying Data	209
Using List-Based Controls	209
Using XML Data	217
Implementing List Selection	229
Connecting to Search Results	231
Dragging and Dropping in Lists	234
Creating Custom Item Renderers	236
Working with External Data Services	240
Summary	241

Chapter 12

Controlling Visibility and Navigation	243
Controlling Visibility	244
Navigation Components	244
Creating a Photo Gallery Application	256
Summary	268

Chapter 13

Working with View States	269
Scenarios for States	269
Managing States in Design Mode	271
Making a Login/Registration Form	275
Applying States to the Search Application	279
Summary	284

Chapter 14

Applying Effects, Transitions, and Filters	285
Effects	286
Transitions	297
Filters	306
Summary	312

Chapter 15

Styling and Skinning	313
Inline Style Assignments	314
Style Blocks and CSS	316
External CSS	320
Skinning	326
Summary	343

Chapter 16

Making Data Dynamic: Linking Flex to the Server ...	345
Some Background Information	345
The ContactManager Application	347
Linking ContactManager to the Server Using the HTTPService Class	356
Summary	364

Chapter 17

Deploying Flex Applications	365
Deploying to the Web	365
Deploying to the Desktop	376
Summary	386

Chapter 18

What Comes Next?	387
Third-Party APIs	388
Print Resources	390
Online Resources	391
Certification	393
Enfin	394

Appendix A

Creating a Development Environment.....	395
Use WAMP (Windows) or MAMP (Mac OS)	395
Add PHP Development Tools (PDT) to a Flash Builder Installation.....	398
Summary	407

Appendix B

MySQL Basics.....	409
Language Elements and Syntax	410
MySQL Statements	411
Creating a Database with phpMyAdmin	414
Summary	419

Appendix C

PHP Basics	421
Language Elements and Syntax	421
The PHP Scripts	424
Summary	429

Appendix D

Compiling Flex Applications on Linux Using the Command Line..... 431

Install Flash Player 10	431
Install Java	433
Download the Flex 4 SDK.....	434
Create a Project Folder Structure	435
Add an MXML File	436
Add Environment Variables	436
Tweak the Project Configuration File	437
Create a Reusable Compiler Script in Bash.....	439
Compile and Test	440
Summary	440

Index..... 441

PREFACE

“Gentlemen, I am tormented by questions; answer them for me.”

—Fyodor Dostoyevsky

Something motivated you to pick up this book. Perhaps you want to start programming, but you're not sure where to begin. If so, you're in the right place, because Adobe Flex allows you to develop programs that run on the Web as well as the desktop. Or maybe you know a little about web development, but you'd like to start creating Web 2.0 applications? Because Flex applications run in Flash Player, Flex is a great framework for creating graphically rich websites, media players, mapping applications, and the like. Or better yet, maybe you already have some basic programming experience and you just want to learn a new language. If you fall into any of these categories, this book should meet your expectations.

Web 2.0 and the Rich Internet Application (RIA)

Web 2.0 refers to a modernizing trend observed in web design in recent years. Websites characterized as Web 2.0 engage their visitors with interactive content. Wikipedia identifies Web 2.0 sites as “social-networking sites, blogs, wikis, video-sharing sites, hosted services, web applications, mashups, and folksonomies.” The potential for user-driven content is another significant trait of Web 2.0. Whether or not you've encountered the term before, it stands to reason that you're acquainted with the Web 2.0 interface.

The **Rich Internet Application** (RIA) is a reaction to opportunities created by advanced web development technologies. Quick site responsiveness to user interaction is the defining trait of a RIA—in essence, it's user interaction without a full-page refresh. While RIAs refer to Internet applications that emulate traditional desktop applications, this interpretation can be somewhat misleading, as nothing says RIAs have to occupy the full screen. A RIA may be a simple widget, sized to fit a small content area within a larger page.

NOTE

Middleware and database combinations could include ASP.NET and Microsoft's SQL Server, PHP and MySQL, ColdFusion and Oracle, and many others. Plus, these combinations aren't inseparable from one another. Many mixtures work quite well.

Perhaps you're thinking, "What can I do with Flex, and where can it take me?" The short answer is this: "You can do a lot with Flex, and you'd be surprised where it might take you." At the very least you can make simple widgets for websites or for your desktop computer. But if you're willing to pursue the next level, which means learning a middleware and database combination, you can develop applications to handle a variety of Internet applications and software needs (Figure P-1). Also, Adobe Flex is seeing increasing demand as a framework for enterprise applications, so if you're inclined to stay the course, Flex development could become a career (Figure P-2).

The authors of this book didn't learn to program exclusively in college or by following a direct path. Indeed, our own beginnings inspired this text and its objective of helping new developers learn the craft. As undergraduates, one of us studied anthropology; the other studied history. We first met in middle school, but we started programming for entirely separate reasons. Curiously, almost a dozen years elapsed before we realized how seriously we each pursued programming and software development.

Pure curiosity motivated Alaric to learn to program; to him, computers were just one more thing to master and bend to his will. His first scripts were written as practical jokes to tease his friends. Discovering he had a knack for programming, he eventually dove into advanced projects that helped pay his way through school. Having mastered ActionScript early in his career, Alaric continues to consult on the newest projects using Flash and Flex.

Elijah's path to Flex followed a much different approach. He initially learned basic web design while working in tech support for Internet providers Arkansas.Net and Socket Internet. Later, in graduate school, he focused on geographic information systems (GIS) programming and ultimately created a hydrology analysis toolkit for Missouri State University's Ozarks Environmental and Water Resources Institute (OEWRI). Elijah's university experience taught him the development life cycle, and he eventually landed at VillaGIS, Inc., where he participates in Internet mapping and enterprise application development using Adobe Flex and AIR.

So, where do our experiences merge with yours? It's simple: we think Flex is cool, and we hope you will too. We know you can learn it, and we believe we can help you. And by teaching you what we've already learned, we'll help you get started quickly and avoid some common pitfalls. Our mutual goal is helping you acquire the precious sense of accomplishment that is so essential for new developers.

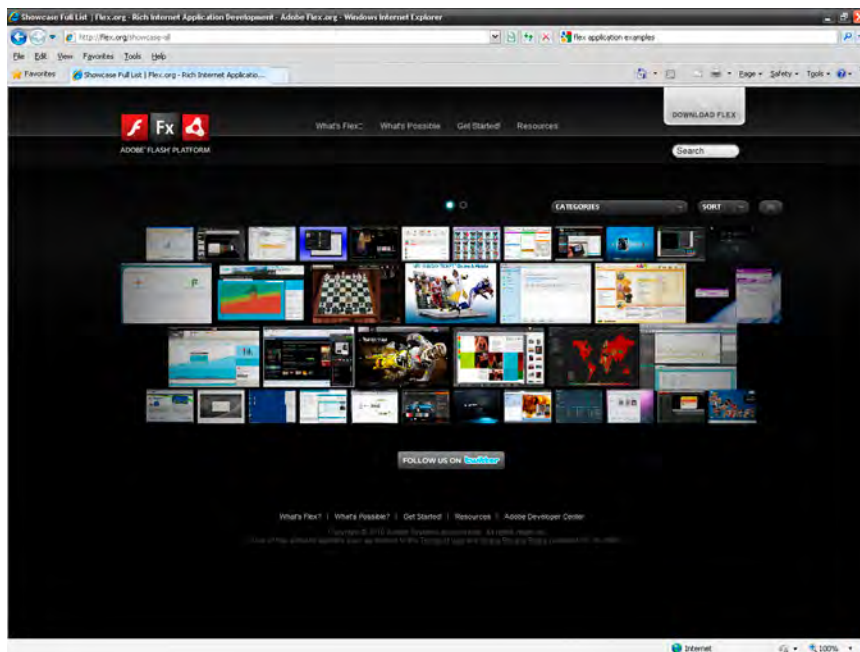


Figure P-1. Flex application showcase (<http://flex.org/showcase-all>)

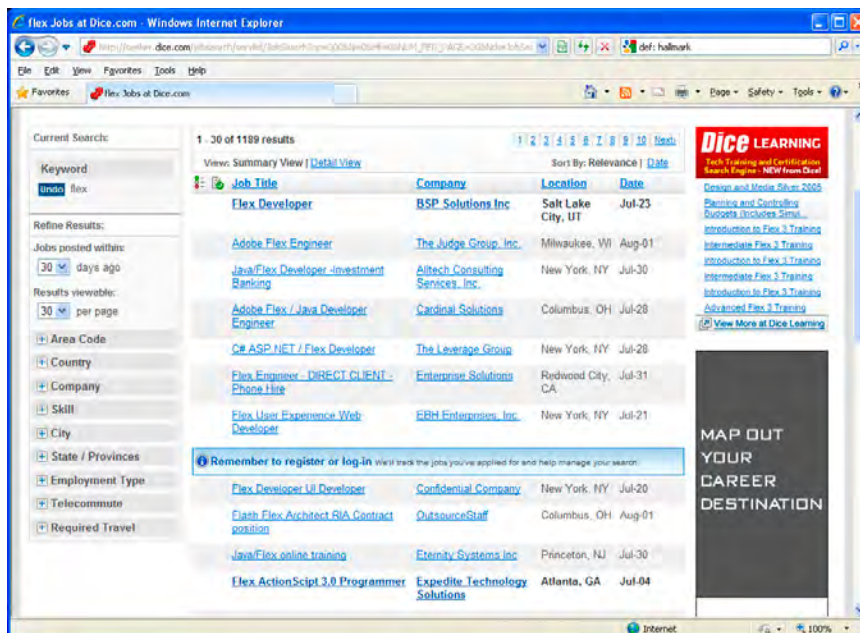


Figure P-2. Searching “flex” at Dice.com (<http://www.dice.com>)

NOTE

Although this is admittedly an introductory text, we'll give you a taste of advanced development in Chapter 16 by exposing you to a simple Flash-PHP-MySQL implementation. It should be enough exposure to help you segue into more specific studies of either PHP and/or MySQL server technologies.

Who This Book Is For

We wrote this book as a guide for anyone learning Flex, but we focused on those completely new to software development. Even if you have no experience with the Flash IDE, web design, or programming in general, we'll make you feel comfortable enough to jump in and tinker with our examples. We will explain basic programming concepts along the way, but this is not a recipe book for advanced projects. Our aim is helping new programmers get started with Flex. Our hope is that each chapter will leave you itching with questions, and that the following chapter will scratch that itch. If we succeed, perhaps you'll step deeper into programming by taking an interest in advanced user interface (UI) development or distributed systems integration.

Flex is a powerful programming environment, but we cover only the nuts and bolts. If you enjoy this technology and want to learn more, there are additional ways to further your studies, including several great books that discuss advanced Flex and ActionScript techniques, as well as blogs, tutorials, and code samples submitted by unsung heroes in the user community.

So, if you're new to programming, we'll get you started with some fun examples, and as always, you're encouraged to extend these examples and make them your own.

How This Book Is Organized

This book is meant to be read cover to cover. Skills are taught progressively, so each chapter builds on the one preceding it. We'll take a hands-on approach, guiding you through key concepts by building applications in stages.

You'll feel comfortable reading each chapter away from your computer, peeking at the code and considering how it affects the applications. Then, once you've absorbed the concept, you can revisit the chapter and construct the example, reinforcing the subject covered by the exercise. Of course, if a topic doesn't interest you, just skip that chapter.

Companion Website

You can download each chapter's source code from the book's companion website, <http://www.learningflex4.com>. So if you skip ahead, experiment, or if your code becomes out of sync with examples in another chapter, just visit the companion website and download code for the appropriate chapter.

We also created a WordPress blog to serve as a forum, and we'll likely post new examples there from time to time; find it at <http://learningflex4.wordpress.com>.

What This Book Covers

Our aim is to give you a step-by-step tutorial through the basic aspects of Flex development. We chose topics designed to empower you and get you started without overwhelming you. We'll familiarize you with the best tool for the job, Adobe's Flash Builder. Then, we'll introduce you to the three languages composing the Flex 4 framework: MXML, ActionScript, and FXG. Finally, once you've learned how to create applications, we'll teach you how to deploy them and share them with others.

We'll have you start by installing Adobe's Flash Builder 4, which is widely considered the best platform for Flex application development. Because Flash Builder is free to try for 60 days, we strongly recommend using it while you learn Flex. Later, once you're comfortable with the framework, you can always revisit your options and experiment with alternative development tools.

After discussing the Flash Builder authoring software, we'll transition into application development by creating the obligatory "Hello World" example. Then, we'll discuss the basics of MXML and ActionScript so you'll understand how these languages work together in Flex. While you cut your teeth on the Flex languages, we'll walk you through building three applications, and in the process you'll learn how to manage event-driven interaction, create dynamic user interfaces, and cycle data through your applications. Of course we'll also discuss "flashy" stuff, including effects, animations, and styles. Toward the end of the book we'll demonstrate a Flex-PHP-MySQL integration, and the final lesson will demonstrate how to deploy projects to the Web or the desktop.

As this is a beginners' book, we will occasionally refer you to other sources we consider helpful with regard to deeper subjects. In fact, if you flip to Chapter 18, you'll find a reasonably dense list of third-party code libraries, books, blog posts, and links to Adobe documentation, all of which might prove useful in your quest.

Typographical Conventions Used in This Book

The following typographical conventions are used in this book:

Plain text

This style is used to describe menu navigation, such as Window→Variables, and keyboard shortcuts, such as Ctrl-C (Windows) or Command-C (Mac OS).

Italic

This styling calls attention to new terms and concepts, much like any instructional text, and is used for emphasis; it's also used to represent filenames and filesystem paths.

NOTE

Just an FYI, Adobe offers Flash Builder for free if you meet any of these criteria:

- *Student, faculty, or staff of an eligible educational institution*
- *Unemployed software developer affected by a recessed economy*
- *Event attendee possessing a special promo code*

This is a very cool campaign. No gimmicks. Check it out for yourself at <https://freeriatools.adobe.com/>.

NOTE

*It's customary to create a "Hello World" example when learning a new programming language. As Scott Rosenberg describes them in *Dreaming in Code* (Crown Publishers), "Hello World" programs are useless but cheerful exercises in ventriloquism; they encourage beginners and speak to the optimist in every programmer. "If I can get it to talk to me, I can get it to do anything!"*

Constant width

Constant width formatting introduces code, either inline among body text or as a block of example code:

```
<s:Button label="Submit"/>
```

Constant width bold

This styling calls your attention to additions or changes within a block of code; it appears most often as a complex example grows:

```
<s:Button label="Edit"/>
```

Constant width italic

This style indicates replacement by specific, user-supplied values.

NOTE

This styling denotes tips, suggestions, or general notes.

WARNING

This styling signals warnings and cautions.

Using the Code Examples

This book is here to help you learn the craft. In general, you're welcome to use examples from this book in your own programs and documentation. You do not need to contact us for permission, unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into a product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the authors' names, the book's title, the publisher, the copyright holder and the ISBN. For example: *Learning Flex 4*, by Alaric Cole and Elijah Robison (O'Reilly). Copyright 2011 Alaric Cole and Elijah Robison, 978-0-596-80563-0.

If you think your use of code examples falls outside fair use or the permission given here, feel free to contact O'Reilly at permissions@oreilly.com.

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

www.oreilly.com/catalog/9780596805630

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

www.oreilly.com

Safari® Books Online



Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at <http://my.safaribooksonline.com>.

Acknowledgments

Like a good film, a good technical book is a product of the combined efforts of lots of dedicated people. While we're the only authors of this book, we couldn't have done it alone.

Thanks from Alaric

Micah Laaker for that initial push and for all the insider tips.

Elijah Robison for taking a chance.

Sharif Zawaideh at <http://globalimagesllc.com> for contributing stunning photos that make the book shine.

Justin Kelly for his reality checks (even though one of them bounced).

Michael Hoch for his un-boss-like understanding and patience when I came to work dreary from all-night writing marathons.

Allen Rabinovich for going above and beyond the title of tech editor in the first edition.

Mom and Dad for being a true mother and father, respectively.

The O'Reilly team for their hard work and support of this book. They're truly the best in the business.

The Adobe Flex team for another solid release.

Thanks from Elijah

Elizabeth Robison, my wife, for being so encouraging and understanding through another exhausting writing campaign. I couldn't have succeeded without your support.

Alaric Cole for being my favorite success story.

Chris Giesey and Jesper Larsen, for helping me with the tough stuff.

Ed Fisher, Curtis Copeland, Lori Giesey, Sean Hills, and everyone in management at VillaGIS—thank you so much for your encouragement throughout this process.

Grant Tomlins, for contributing a few photos to the PhotoGallery application, and for scaring me senseless at least five times a week. Actually, thanks for putting up with such a cranky, exhausted carpooler for so many months.

Matt Sabath for contributing his Linux know-how to Appendix D. Thanks, Matt.

David Baclian (<http://www.davidbenjaminphotography.com/>) and John Van Every for contributing some cool photos to the PhotoGallery application.

Brian Moseley for suggesting several of the quotations introducing the chapters, for being there that time at that place to do that thing, and for being one of the most genuine people I've known.

Dave, Debby, and Aston Quinonez; Billy Ellison; the Brownfield family; Corey Nolan; Dillon Walden; John Wallace; Gabriel Scott Dean; Nick Peña; Matt Moser; Mike Wingate; Brian Fitch; Marcus French; Seth Harrell; Chris Person; Justin Besancon; James Geurtz; Trey Ronne; Jimmy Kribs; my first real coder-friend Mark Ewing; Danqsyle; the curvature of the Earth: and anyone else who encouraged me, inspired me, or otherwise tolerated my absence during the first eight months of 2010.

Mark Morgan, Seth Cartwright, Craig Shipley, and Craig Earl Nelson for initially getting me into computers and cool software.

Kenny Anderson, the first of my friends who ever mentioned “integers” in day-to-day conversation.

Brandon Lee Tate, because I fully agree that Brian May is the most underrated guitarist of all time.

Bob Pavlowsky, Xin Miao, Jun Luo, and William Wedenoja for all the doors you open at Missouri State.

Richard and Margaret Van Every, two generous parents whom I cannot thank enough.

Mike and Linda Gurlen for tolerating my eccentricities while I tested so many waters—and for being fine parents.

Steve Robison for being legendary.

Jodie O'Rourke and Russ Ferguson for providing initial feedback that resulted in a much better book.

Sumita Mukherji and Genevieve d'Entremont for marshaling me through the copyedit process with more speed and precision than Ned Braden and Dean Youngblood combined.

Kristen Borg for transforming the raw manuscript into something resembling a book—and for painstakingly colorizing the code for a couple picky authors.

Mary Treseler, the coolest editor I know, and everyone at O'Reilly Media who made this thing happen.

Finally, the teams developing Open Office, the GIMP, and IrfanView, whose fantastic software was used exclusively when writing the manuscript for this book.

GETTING UP TO SPEED

“Supposing is good, but finding out is better.”

—Mark Twain

This chapter provides a background discussion of Flex, including what it is, what situations it’s suitable for handling, how it compares to other RIA technologies, and when you should *not* use it.

If you’re not presently concerned with background information, feel free to jump ahead to Chapter 2, where we introduce you to the Flash Builder IDE and have you import your first project.

What Is Flex?

Flex 4 is a software authoring framework designed for rapid development of rich, expressive applications. Depending on your goals for a particular project, Flex applications may be developed for the Web, for the desktop, or for both.

Flex isn’t exactly a single computer language. Really, Flex 4 includes three languages—MXML, ActionScript, and FXG—and each satisfies a special purpose. Let’s briefly consider the three languages and see an example of each.

MXML

MXML is typically used to arrange an application’s user interface (UI), the components making up the application’s visible layout (Example 1-1). Similar to a description of a person’s appearance, MXML defines *what* for an application.

IN THIS CHAPTER

What Is Flex?

What About AIR?

Where Flex Fits

Why Use Flex?

How Flex Compares to
Other Technologies

When Not to Use Flex

Summary

What Does MXML Mean?

The acronym MXML was never formally defined. Some think it stands for “Maximum eXperience”. Others say it means “Macromedia XML.” Honestly, though, it’s all conjecture. Refer to it however you like.

NOTE

Don't worry about fully grasping these examples. While we kept them simple and fairly self-explanatory, we just want you to see some basic MXML, ActionScript, and FXG code blocks.

NOTE

ActionScript 3.0 is a modern object-oriented language, and it's much more complex than we're hinting at in this simple description. We'll discuss the tenets of object-oriented programming later in Chapter 5.

NOTE

For more information on FXG graphics, jump to Chapter 14 and read the sidebar titled "FXG Graphics" on page 288. It won't hurt to read it out of sequence.

Example 1-1. MXML: A VGroup container with two TextInput controls and a Button; the Button's "click" event calls an ActionScript function

```
<s:VGroup>
  <s:TextInput id="inputTI"/>
  <s:Button label="Refresh" click="refreshOutput()"/>
  <s:TextInput id="outputTI"/>
</s:VGroup>
```

ActionScript

For now, think of ActionScript as handling an application's *reactive* qualities. Use ActionScript to provide step-by-step instructions when user interaction occurs or when something needs to happen (see Example 1-2). ActionScript describes *how*.

Example 1-2. This ActionScript function takes the text value of one TextInput control and assigns it to the text value of another TextInput

```
private function refreshOutput():void{
  outputTI.text = inputTI.text;
  inputTI.text = "";
}
```

Flash XML Graphics (FXG)

Stated simply, FXG is an XML-based standard for sharing graphics between various Adobe products. FXG graphics can be created individually by a programmer, and simple graphics often are (see Example 1-3). However, the real benefit of FXG graphics is realized by creating complex graphics in design software such as Adobe Illustrator or Photoshop, exporting them as FXG files, and then importing them into Flex as reusable graphics objects.

Example 1-3. An FXG Rectangle graphic with both a fill and a stroke creating the background and border for an MXML Group container

```
<s:Group id="container" horizontalCenter="0" verticalCenter="0">

  <s:Rect height="100%" width="100%">
    <s:fill>
      <s:SolidColor color="#209910"/>
    </s:fill>
    <s:stroke>
      <s:SolidColorStroke color="#000000" weight="2"/>
    </s:stroke>
  </s:Rect>

</s:Group>
```

Flex Is a Modern, Hybrid Framework

Flex will feel natural to web developers familiar with Hypertext Markup Language (HTML) and JavaScript. Although different architecturally, the similarities should make it easy for experienced developers to get started. MXML is a markup language, and ActionScript is a scripting language. Like modern web design, Flex combines markup and scripting for application development.

Flex Is Flash

Flex applications, like other Flash content, are deployed as SWF files (pronounced “swiff”). SWF files are compiled projects that Flash Player reads and renders onscreen, often in a browser. With Flex, you can create applications that are small enough to download quickly and that will, with few exceptions, look and act the same on any computer or operating system (Figure 1-1).

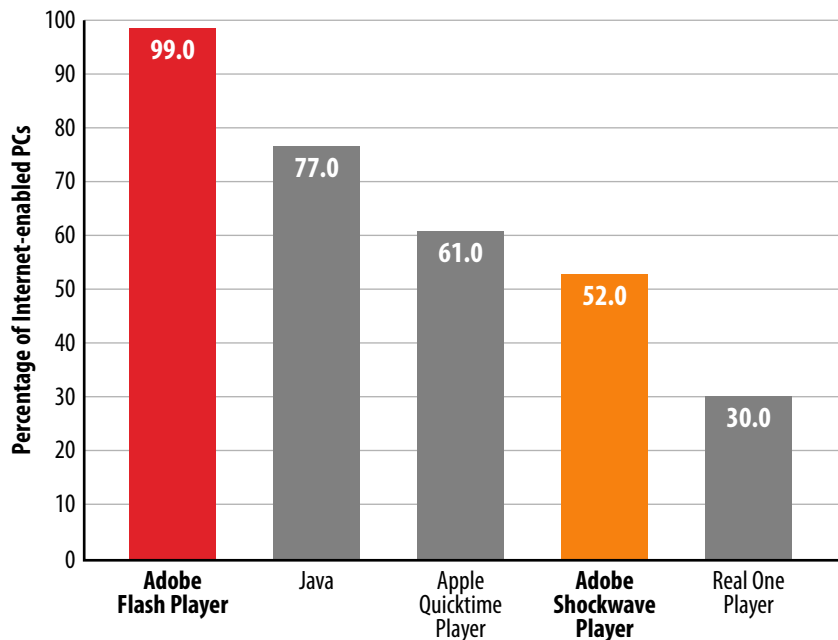


Figure 1-1. Flash Player reach

What Does Flex Look Like?

Although there is a default appearance, the look and feel of a Flex application is not set in stone. Flex doesn't have to look like Windows or Mac or anything else; rather, it's easily styled and fully skinnable.

In Flex development, **styling** refers to the assignment of basic colors, font characteristics, padding, and the like using Cascading Style Sheets (CSS). Stylesheets are simple, so style values defined in CSS can be reworked quickly.

On the other hand, **skinning** implies broad change of a component's look, shape, or behavior. In the Flex 4 designer-developer workflow, skinning often involves collaboration between a graphic designer and a developer to create a unique, “branded” look for an application.

Finally, there are **themes**. If you do some quick digging on the Web, you can find a number of themes that let you rapidly change the entire look of your application with one fast action. With the right code, you could even let the users of your application select a preferred theme!

What's the Flash Platform?

The Flash Platform is an overall term referring to several technologies composing the Flash “ecosystem,” which Adobe divides into five categories: tools, framework, servers, services, and clients.

In the context of this book, we'll be using Flash Builder—the tool—to develop applications in Flex—the framework—that will be handled at runtime by either the Flash Player or the AIR client.

Other elements of the Flash Platform include Flash Professional, Flash Catalyst, Flash Media Server, LiveCycle, and BlazeDS. For more information, check out Adobe's website: <http://www.adobe.com/flashplatform>.

Flex Is the Flex SDK

The Flex Software Development Kit (SDK) consists of an extensive library of UI components, a compiler, and documentation tools that facilitate development. To illustrate, creating a Button component no longer requires writing several lines of ActionScript code or dealing with timeline animation; Flex developers can simply type `<s:Button/>` or use Flash Builder's Design mode to drag and drop a Button wherever they want in an application.

Thanks to the SDK tools, Flex developers can create applications using any text editor and the free compiler, but Flash Builder—Adobe's luxury IDE—certainly provides a more comfortable development experience. In other words, it's not necessary to create Flex applications in Flash Builder, but it helps. Because Adobe provides a 60-day trial of Flash Builder, we recommend using it while you learn Flex. Once you're comfortable programming MXML and ActionScript, you can always switch to another IDE and/or command-line compiling.

What About AIR?

Adobe Integrated Runtime (AIR) is the solution for bringing Flash and other web-based content to the desktop. With all the benefits of a RIA, why would anyone want to do this? Mainly, browsers have limitations—they don't support drag-and-drop from the desktop, they have security restrictions, and users can't access web applications when they don't have an Internet connection. Also, many users like having an application in their Start menu or Dock for quick access.

Here's another reason AIR shines: it opens up desktop development to the seasoned web developer. Because of the broad demand for web development, many programmers are already familiar with HTML, JavaScript, and Flash, as well as the architecture of a complex website. With Adobe AIR, there's no need to learn C# or Java just to create a standalone application; rather, familiar skills can be used to start creating desktop applications with a shorter learning curve. This is especially significant if you intend to make both web and desktop applications.

Finally—and possibly the most compelling reason to use AIR—Flash applications are platform agnostic, which means your desktop application won't be chained to a particular operating system. You won't have to worry about developing for Windows, Mac, or Linux; you simply write a program in Flex or JavaScript, and anyone can run it as an AIR application.

Where Flex Fits

The phrase *Rich Internet Application* (RIA) was coined by Macromedia (now Adobe) in 2002 to describe a trend toward more expressive web applications.

In the beginning, HTML documents on the Web were just that: documents. They were text and, later, images and multimedia. This original client-server paradigm meant a user, by typing a URL in his browser, requested a document. With the Web being so far-reaching, savvy developers quickly learned to create server-based applications, programs anyone could access while online. Think of all the web forms you've filled out, where you type your name and address and hit the submit button. After a few moments, you're greeted with an entirely new page telling you the form was submitted (or that you have errors to fix). This interaction describes the client-server model, where a "thin" client (a browser) gathers content and sends it back to a server for processing. Besides being slow, this approach wasted client resources at the expense of the server.

The advent of JavaScript introduced a new potential—offloading work to the client. For instance, when configuring an item in an online store, it used to be necessary to send all calculations, such as shipping or sales tax, back to the server. With scripting, it became possible to calculate that information right on the client machine and update the page according to user interaction. This new approach marked the beginning of the "thick" client, which describes client-server relationships where the client shares processing responsibilities with the server.

However, for developers wanting animations, transitions, and a richer experience, there was a need to load data without refreshing the entire page. Before Ajax entered the scene, the answer for many was Flash.

Flash got its start as a solution for rendering animations and multimedia on the Web. As it matured, more and more interactive elements were added, and some people learned how to create Flash ads and games.

Because Flash was quick, lightweight, and naturally rich, other developers started using it in complex applications such as data visualizers, product catalogs, and photo viewers. Because the Flash IDE was an animation tool, creating complex interaction was often difficult and a bit messy. The introduction of Flash Professional in 2003 improved the process by adding key features such as reusable components and data connectors, but it still left a lot to be desired, especially for enterprise developments and projects with large teams.

Enter Flex. Building upon the power of Flash Player, Flex made it easy to develop rich user interfaces. With a more developer-centric model, it was easier for Java programmers and others to jump on board and start developing without the "what is this timeline?" confusion. Even better, the markup-based language made it easy to read and share code, which has definitely contributed to its growing success.

NOTE

A number of image editing utilities exist as web applications, and you can find several with a quick search. Most of the online editors we found appeared to be built with Flex (they ran in Flash Player), and they had decent user interfaces, but the FXG Editor by 7jigen Labs stood out as our favorite, not because it had the most graceful UI or the greatest features, but because it provided such a rare, sought-after service—online FXG editing! If you don't have Adobe Illustrator but want to create FXG graphics, this is the only free option we've found.

You can access the FXG Editor by 7jigen Labs at <http://fxgeditor.7jigen.net/>.

They also make it available as an AIR application: <http://labs.7jigen.net/2010/05/15/fxg-editor-air-app/>.

What's even better? 7jigen made their source code available. Check it out someday when you're feeling ambitious: <http://www.libspark.org/wiki/seven/fxgeditor/en>.

Why Use Flex?

“Why use Flex?” Did you ask yourself this question when you picked up this book? Flex was created to simplify development of web applications for Flash Player, but it has evolved into a broadly useful framework that lends itself to both web and desktop application development.

Flex Is for Applications

The Flex framework was built to streamline application development, and as such, it includes a number of robust, customizable components that make designing and developing an application relatively easy. Flex programs can run in a web browser or be deployed as AIR applications, so Flex is ideal for writing an application once and delivering it to anyone.

Web Applications Defined

A desktop application is any software you use on a personal computer, such as a web browser or a word processor. In contrast, a web application is any application you access through a web browser.

Some applications may dually exist on the desktop as well as the Web. Email applications provide a familiar example. The desktop email client makes it easy to read archived email whenever you please, even when you're offline, but web access creates the advantage of sending and receiving email from any Internet-ready computer, such as those in a classroom, a workplace, or a friend's house.

Now imagine if the desktop email client and the web portal were nearly identical, having the same look and feel, and even the same code base! Flex makes that possible.

For Easy Interactivity

Flex makes it easy to create a high level of interactivity. It comes with support for data binding, a smooth event architecture, and a set of components with built-in methods for providing user feedback. Add to this the ability to quickly create beautiful effects and transitions, and it's easy to see why Flex is getting so much attention from new developers.

For Development Speed

There is no faster way to create such rich, interactive, uniquely skinned applications. Although everything you can do in Flex can be done in the Flash IDE, development in Flex reduces the process to a fraction of the time it takes in Flash. And with Flash Builder, development is faster still.

For Speed All Around

Flex components are built in ActionScript 3.0, the latest version of the programming language for Flash Player, and Flash Player 10 was introduced in tandem with the release of Flex 4. Improved performance was one of the goals for Flash Player 10, and it shows. Although you should always make an effort to optimize your programs, you can feel confident that Flex 4 and Flash Player 10 “have your back” when it comes to handling graphically intense routines.

Because It’s Clean

Flex supports separation of content and design by allowing external styling of applications. Because of this, you can quickly restyle an application by referencing a fresh stylesheet or by dropping in one of the many freely available Flex themes.

Proponents of the Model-View-Controller (MVC) design pattern will find that Flex supports that architecture as well. Free libraries are available, such as the Cairngorm framework, which simplifies implementing MVC.

Because It’s Free

Although you have to pay for a full copy of Flash Builder, the Flex framework is completely free. Therefore, you have the option of creating and editing code in your favorite text editor and compiling it using the free command-line compiler without paying a dime.

The scope of this book is for beginners, and because Flash Builder makes it easy to get started, we’ll discuss development, compiling, and debugging using Flash Builder. Luckily, Adobe offers a free 60-day trial, so there’s no reason not to use Flash Builder while you’re learning Flex: <http://www.adobe.com/products/flex>.

Because It’s Open

Flex is also open source. This means all the component code is yours for the looking (and using and reusing in most cases). As a new developer, having access to the underlying code helps you learn. Open source also means Flex is yours. You can modify and improve the code as well as submit your modifications for inclusion in future versions.

Because it's open source, a community has grown around extending and improving Flex. Scores of freely available advanced components have been created to extend the base set. As such, it's usually possible to find special components, as well as tutorials explaining how they're created from scratch.

NOTE

For more information on the Cairngorm framework, start at <http://opensource.adobe.com/wiki/display/cairngorm/Cairngorm>.

While we’re on the subject of Flex frameworks, the Swiz framework is becoming increasingly popular. Its torch-bearers tout its lightweight simplicity, lack of so-called “boilerplate” code, and techniques for creating components to handle repetitive situations. If you can’t tell, we’re describing advanced Flex development, but in the spirit of learning, we recommend that you to look into it someday: <http://swizframework.org/>.

NOTE

Only standard components are available to you in the free SDK. Fortunately, most Flex components are standard components. In comparison, advanced components, such as the data visualization (i.e., charting) components, require a Flash Builder license.

For Data (and Fast)

Flex offers built-in support for XML and Java objects, simplifying data exchange using these approaches. Flex also supports Action Message Format (AMF). With Java and ColdFusion servers, you can transmit compressed binary data over the wire to your Flex application, making data submission and retrieval much faster than text-based solutions. And for the freeware folks, you can also combine Flex with AMFPHP to send and receive serialized ActionScript objects to and from server-side PHP scripts.

Because It's Beautiful

Although Flex comes with a default theme that may suit your needs, an application's look and feel is limited only by your imagination. Plus, because styles can be established using Cascading Style Sheets, you can change the look of your program with a single line of code.

Flex provides a robust set of UI controls right out of the box, and with the wide variety of third-party components, you can create any interface you desire. With Flex Charting, you have a nice set of data visualization tools at your disposal. You can choose from bar charts, pie charts, high-low-open-close charts—you name it. Moreover, thanks to the ease of development in Flex, the number of third-party data visualization components is growing every day.

How Flex Compares to Other Technologies

Flex is a hybrid technology, taking the best elements from modern programming languages while incorporating standards such as XML and (CSS). Because of this, Flex bears a familiar resemblance to several technologies.

Flash

Like Flash, Flex applications run in Flash Player. However, aside from having the same underlying scripting language—ActionScript—these two technologies have some differences. At its core, Flash was created to run Timeline-based animations. Development features were added later. On the other hand, Flex was designed from the ground up to simplify application development. Users of Flash who have dealt only with the Timeline or simple scripting may find Flex a bit overwhelming at first, whereas Java or C developers will feel more at home.

There's nothing you can do in Flex that you can't do in Flash—technically, that is. It's possible to develop a great application using just the Flash IDE. However, there's always a right tool for a job, and Flex was engineered to help you create applications. Specifically, Flex supports data management, simple styling using Cascading Style Sheets, and an improved designer-developer workflow for advanced component skinning. Plus, Flex makes it easy to add rich animations and graphical effects to your programs.

However, Flex is *not* a drawing framework or animation toolkit. So if you're looking to create movies or animated cartoons, the Flash IDE is the right tool for the job.

C Languages

Flash Builder is an IDE similar to Visual Studio, and with it you can write code, compile, debug, and deploy applications from a single piece of software. Flex's MXML markup language may be unfamiliar at first, but it's intuitive, so you can expect to learn it rapidly. And while everyone is entitled to her own opinion, some might say it's difficult to beat MXML when it comes to designing an interface.

Though Flex is based on a language different from C++, Objective-C, and so on, developers familiar with scripting languages should feel comfortable learning ActionScript. Because ActionScript is a class-based, object-oriented language, seasoned C++ and .NET developers should adapt quickly to Flex. However, C# developers may find the easiest transition, because that language shares a number of commonalities.

Java/Java FX

Flex is similar to Java and the Java Swing platform. For one, ActionScript is similarly structured—it inherits the concept of packages, and its syntax is nearly identical. MXML poses the biggest difference, but as you'll discover, it's easy to learn. Because Flash Builder is built on Eclipse, many Java programmers will already be comfortable using the IDE.

Java, like Flex, also allows for application deployment to either the Web or the desktop. However, the ubiquity and small size of Flash Player compared to the Java SDK makes Flex applications available to a wider audience. A lot of Java developers are learning Flex, and a lot of projects that might have been built in Java are moving to Flex.

HTML/JavaScript/Ajax

Flex was built after the web explosion, and for this reason its design features a number of commonalities to traditional web development. Most notably, Flex couples a tag-based language (MXML) with an ECMA-compliant language (ActionScript). Web programmers with knowledge of HTML and XML

A Note About the Term "Flash"

In this book, we'll frequently mention "Flash," a term that for many is tough to define. Depending upon the context, Flash can mean many things. For instance, Flash may refer to the Flash Integrated Development Environment (IDE), the animation and development tool that started it all. In other contexts, Flash may refer to the content you find on the Web—the animations, advertisements, or applications, which are compiled SWF files running inside a browser. In yet another capacity, Flash is an umbrella term describing the technology built upon Flash Player, the plug-in that makes it possible to view all this stuff on your computer.

Out in the wild, you'll hear "Flash" used in various contexts, but in this book, we'll try to keep things consistent. When we say "Flash," we're referring to the underlying technology requiring Flash Player. When we discuss the development software, we'll say the "Flash IDE" or the "Flash authoring tool."

will quickly absorb MXML, and because JavaScript syntax is so similar to ActionScript, web developers proficient in JavaScript will transition easily to ActionScript. Behind the scenes, MXML and ActionScript have a different relationship to one another than HTML and JavaScript, but on the surface, their interaction will make sense to most traditional web developers.

Ajax (Asynchronous JavaScript and XML) and Flex have several similarities, and developers of either technology can be downright religious when it comes to identifying the favorite. Both Flex and Ajax allow developers to create responsive web applications that avoid clumsy, full-page reloads. Regarding their differences, Ajax is “leaner” than Flex, as it doesn’t require a framework download before the application is available to the user. However, strict browser settings can cripple Ajax applications, leaving users confused as to why the application isn’t working. Like with Flex, several third-party Ajax libraries have emerged, and the Ajax libraries tend to be more mature than their Flex counterparts; the Google Maps API comes to mind in this respect. In Flex’s favor, MXML/ActionScript code is arguably easier to write and maintain than Ajax code, and the Flash Player provides superior support for graphics and animation, including charting and data visualization. However, Ajax is more suitable for applications that need to load quickly, and this is especially true if the application will be graphically lightweight.

Silverlight/XAML

Silverlight is Microsoft’s answer to Flex; in fact, it shares the XML-based markup paradigm coupled with a choice of programming languages. In this way, knowing Silverlight will definitely help in learning Flex, and vice versa. With Flex, thanks to the Flash plug-in, you can expect your programs to perform almost identically anywhere they’re deployed. Silverlight also attempts to offer cross-platform compatibility using a browser plug-in, and although the Silverlight plug-in is larger, its reach is growing. Like Flex, Silverlight offers several advanced controls and layout containers; however, because Flex is open source, the number of third-party Flex components has the potential to increase at a greater pace.

OpenLaszlo

OpenLaszlo is a popular open source framework using Scalable Vector Graphics (SVG) and XML/JavaScript to build RIAs. As such, developers comfortable with this framework will surely make an easy adjustment to MXML and ActionScript.

When Not to Use Flex

Flex is a great technology, and many will find it solves issues that have plagued past development. However, it's not for everyone.

If you're looking to create simple animations and don't want to write any code, a Timeline-based animation utility might be a better choice. Because Flex requires a component framework, its applications often result in larger file sizes than custom Flash or ActionScript-only applications. However, in many cases, a slight increase in size may be worth the reduced development time. If you don't want simple widgets or small applications to be weighed down by the framework, there are some solutions. For instance, it's not necessary to use the Flex components or even MXML to create Flex widgets. It's completely possible to create an ActionScript-only project and compile it using the Flex compiler, and if this is your intention, Flash Builder is a great tool for such projects.

If your project requires a lot of richly formatted text and basic user interactions, you might be better off using HTML/Ajax. While Flex has decent support for HTML, it may be insufficient for some projects. However, if you want to reserve the option of deploying to the desktop via Adobe AIR, Flex might become the better choice, as AIR has native support for the full gamut of HTML. If you only intend to develop websites with loads of text, just use HTML.

Summary

Hopefully, now you have a better idea of what Flex is all about. While Flex was created to simplify web application development for Flash Player, Flex has become a key player in the world of rich applications for the Web and the desktop. Since a Flex UI can be created rapidly using MXML and CSS, and because an application's appearance and scripted functionality are easily separated, Flex helps simplify exchanges between designers and developers working collaboratively. You've seen how Flex compares to other technologies, and you've learned what Flex is good for as well as when it's not the best choice. If Flex is the tool you're looking to learn, the following chapters will get you rolling. The next chapter jumps right into the basics of using Flex and Flash Builder.

How Do I Know It's a Flex Application?

With the explosion of Web 2.0 and the RIA, the lines have blurred between what's possible in HTML and what's reserved for Flash. Just a few years ago, if you saw a fancy transition or animation, you could be sure it was Flash. Now, though, it's tough to tell just by looking.

One trick to test whether a site or widget was created using Flex is to right-click (Control-click on a Mac) some area of the application. If the content is Flash or Flex, a context menu will appear noting "About Adobe Flash Player" at the bottom. Although this test can guarantee content is running in Flash Player, it doesn't confirm it was built using Flex. For that, unfortunately, there's no sure method, because folks have built some pretty sophisticated applications using just the Flash IDE. However, once you get a feel for a few Flex applications and become familiar with the most common components, you can generally spot a Flex application simply by seeing it and interacting with it.

SETTING UP YOUR ENVIRONMENT

“Give me six hours to chop down a tree, and I will spend the first four sharpening the axe.”

—Abraham Lincoln

Adobe’s Flex 4 SDK is free, and with it, you can use any text editor to create Flex applications. However, if you adopt this approach you’ll be working without code completion, and you’ll need to compile applications using the SDK’s command-line compiler. This doesn’t exactly describe a course of least resistance. That said, Adobe’s Flash Builder 4 is designed to simplify Flex development. Regarded as the premier IDE for Flex development, Flash Builder provides a customizable code editor, code completion, informative framework documentation, automated compiling, debugging tools, and utilities that assist team-based development.

Almost anyone who’s familiar with Flex and who has used another editor to develop Flex applications will tell you Flash Builder is the IDE of choice. Of course, there are other options available to you, which we discuss next, but this book will introduce Flex development using Flash Builder.

If you don’t have Flash Builder yet, it’s easy to download it from Adobe at <http://www.adobe.com/products/flex/>. If you prefer to use your own editor and compile via the command line or other alternatives, you can download just the Flex SDK. However, we recommend starting with a copy of Flash Builder because Adobe will give you 60 days to try it with no limits. (Actually, they’ll put a watermark in any chart you create, but besides that, all the features will be there.)

You have two options when downloading and/or buying Flash Builder: Adobe’s standalone Flash Builder installation or their Eclipse plug-in. We discuss these options later in the section titled “Flex Flavors” on page 16.

IN THIS CHAPTER

Using Alternatives to Flash Builder

Introducing Flash Builder and Eclipse

Running Your First Application

Creating a New Flex Project
Summary

NOTE

A Software Development Kit (SDK) is a collection of library code, sample files, documentation, and/or utilities that enable development in a particular language or for a special purpose.

NOTE

An Integrated Development Environment (IDE) is an application designed to help developers write programs. A basic IDE should provide code editing, compiling, and debugging tools, whereas an advanced IDE might offer additional luxuries such as code completion, version control, bug tracking, and project management utilities. For more information, see the sidebar titled “More About the IDE” on page 18.

NOTE

If you want to try alternative editors, you'll likely need the Flex 4 SDK. Visit the following site to get the Flex 4 SDK directly from Adobe: <http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4>.

NOTE

Shortly before going to press, we found a blog post that explains how to integrate the Flex 4 SDK into Eclipse; see <http://www.seanhsmith.com/2010/03/29/flex-for-free-setting-up-the-flex-4-sdk-with-eclipse-ide/>. It's worth your while to check it out, although it seems you'll still be without code completion.

Using Alternatives to Flash Builder

If your copy of Flash Builder has expired, or if you're just being stubborn, there are alternatives available. Since Flex's MXML is an XML-based language, you'll benefit from an editor that offers XML markup, such as tab indent and color-coded syntax. Moreover, to avoid undue frustration, any editor that can automate Flex compiling should be considered ahead of those that don't. With this advice in mind, check out this list of alternative editors that may interest you.

FDT (Windows, Mac OS)

FDT is a professional IDE for Actionscript and Flex projects, and some might say it's superior to Flash Builder. As always, you're welcome to form your own opinion, as Powerflasher Solutions—the makers of FDT—allow you to evaluate their product for 30 days. Visit <http://www.fdt.powerflasher.com/>.

FlashDevelop (Windows)

FlashDevelop is a popular open source IDE for ActionScript developers, and you can use it to edit and compile Flex applications as well. Like Eclipse, FlashDevelop is free, but since it offers code completion and automated compiling (see the sidebar “Configure FlashDevelop to Use the Flex 4 SDK” on page 15), it gets our vote as the best free alternative. Visit <http://osflash.org/flashdevelop>.

Eclipse (Windows, Mac OS, Linux)

Eclipse is a free, open source IDE with an impressive following in the Java community. You can use Eclipse to develop in many different languages, and even better, you can configure Eclipse to compile Flex applications using the Flex 4 SDK. Visit <http://www.eclipse.org/>.

TextMate (Mac OS)

TextMate is a great text editor with built-in support for ActionScript, but if you try it, look for a Flex “bundle” that will simplify working with MXML. Visit <http://macromates.com/>.

Notepad++ (Windows)

Notepad++ is a neat editor to have in your arsenal. Besides support for HTML and XML, one of its advertised features is the ability to implement third-party code completion utilities. See <http://notepad-plus.sourceforge.net/uk/site.htm>.

TextPad (Windows)

Touted as a powerful editor that's easy to get up and running, TextPad is a good choice for coding MXML by hand in Windows. Visit <http://www.textpad.com/>.

Configure FlashDevelop to Use the Flex 4 SDK

FlashDevelop is a reasonable alternative to Flash Builder, especially for a free IDE. Unfortunately, FlashDevelop is available only for Windows, but if you're a Windows user, it's easy to configure FlashDevelop to use the Flex 4 SDK, and doing so enables both code completion and automated compiling (Figure 2-1).

Assuming you've installed FlashDevelop (we used version 3.2.2 RTM) and downloaded/unzipped the Flex 4 SDK (we unzipped our SDK to *C:\flex4sdk*), point FlashDevelop at your SDK folder by selecting Project→Properties, and then clicking the Compiler Options tab.

Under the Compiler Options tab, locate the field "Custom Path to Flex SDK" and assign a value pointing to your unzipped Flex 4 SDK, for example, *C:\flex4sdk*.

That will be enough to enable code completion and compiling, but you'll need to do one more thing to make sure FlashDevelop properly launches a compiled application. The easiest solution is to configure FlashDevelop to call Flash Player, which you can do by selecting Project→Properties, and then clicking the Output tab.

Under the Output tab, locate the "Test Movie" option at the bottom of the dialog and select "Play in popup."

That's all there is to it! Now you can use this free tool to create and compile Flex 4 applications.

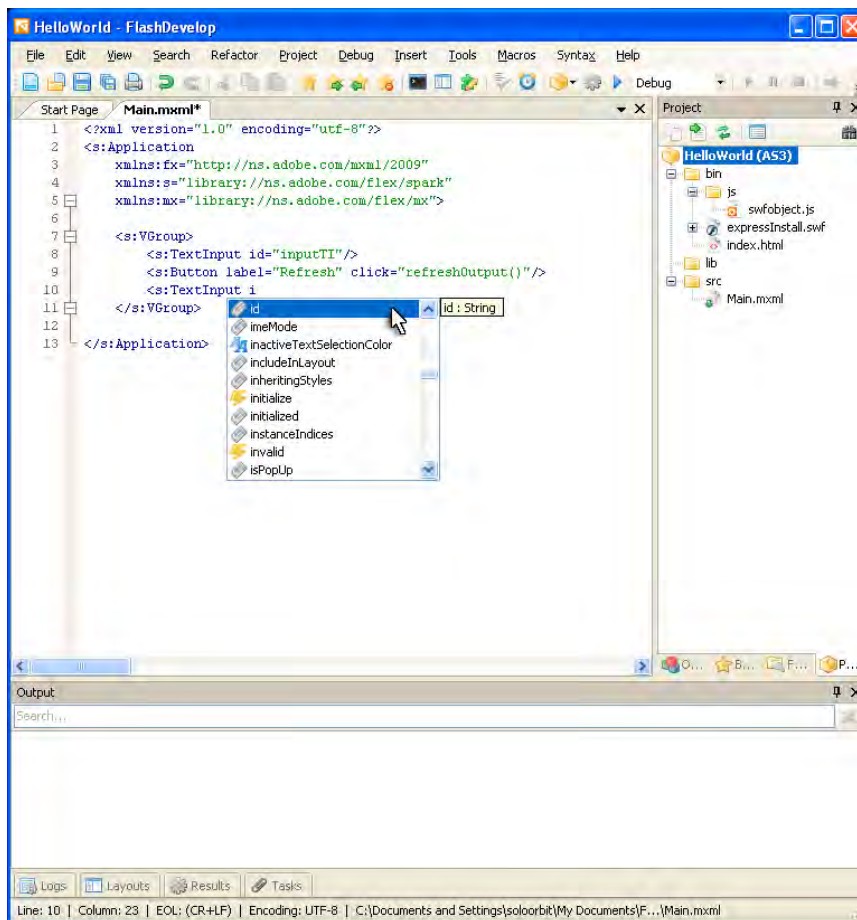


Figure 2-1. Creating a Flex 4 application with code completion using FlashDevelop and the Flex 4 SDK

NOTE

For more information about using the command line to compile Flex code, check out Adobe's documentation on the Flex 4 compiler API at http://help.adobe.com/en_US/Flex/4.0/CompilerAPI/flex_4.0_compilerapi.pdf.

While we're on the subject of Adobe documentation, we'd be remiss if we didn't point you toward the Flex Developer Center at <http://www.adobe.com/devnet/flex/?view=documentation>.

It's loaded with various resources to help you learn and grow as a Flex developer. Check it out. Bookmark it, even.

After using a bare-bones editor to write code, you'll still need to compile that code before you have a functioning application. Because Flex is an open platform, new compiling options become available every day, so a quick search might reveal just what you need. Realize that most third-party solutions will implement the free command-line compiler, which you can interface using either a command prompt (Windows) or terminal window (Mac OS, Linux). By the way, the command-line compiler does provide some debugging support, so you won't be flying blind; however, the editor-and-compiler approach is definitely not for the weak-willed.

Introducing Flash Builder and Eclipse

Once you have a copy of Flash Builder, go through the program to familiarize yourself with the most important features. Flash Builder is built on the popular and open source Eclipse IDE. Eclipse is powerful, but it might not be the most beautiful or user-friendly program ever developed, so we'll hold your hand through the process of discovering what's what.

Flex Flavors

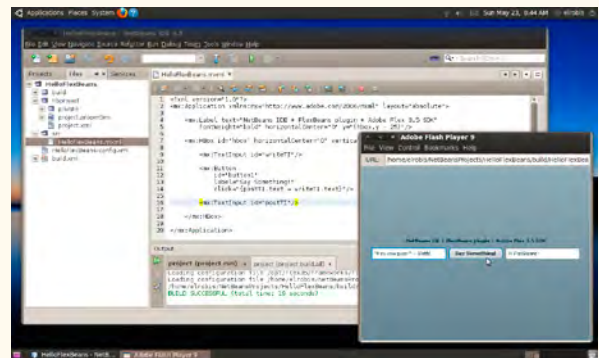
Flash Builder comes in two flavors: a standalone installation or an Eclipse plug-in. What's the difference, you ask? Well, branding.

If you're already familiar with the Eclipse IDE and you develop in several languages and frameworks, Adobe's Flash Builder plug-in for Eclipse is more appropriate for your needs. However, if Flex development is your primary interest and/or you're unfamiliar with Eclipse, be good to yourself and get Adobe's standalone installation. The standalone installation comes with several great features, such as a Flex icon for your Dock/Start menu, a Flex splash screen, and...well, you get the idea. We'll use the standalone version in the book examples, so if you're using the Eclipse plug-in, there will be situations when the screen looks different and menu navigation isn't identical.

There's no indicator that Linux will see a fully qualified Flex 4 IDE from Adobe any time soon.

That doesn't mean Linux users are entirely without options. Other IDE utilities can be configured for developing Flex 4 against the free SDK, but presently, we're not aware of any tools that offer code completion. If you know of something, please strike up a discussion on the companion website's WordPress forum at <http://learningflex4.wordpress.com/>.

Finally, you might find an occasional blog post from a grinder tearing it up with the free command-line compiler. Surprisingly, we had more luck creating a simple Flex 4 application using Ubuntu 10.04, *gedit* (a built-in editor), the Flex 4 SDK, and Adobe's command-line compiler (Figure 2-4). If you're interested in trying this approach, check out Appendix D, *Compiling Flex Applications on Linux Using the Command Line*, which was written by Matthew Sabbath, one of Elijah's coworkers at VillaGIS.

[illegible]

Chapter 2, Setting Up Your Environment

More About the IDE

Again, IDE stands for Integrated Development Environment, and the term describes software made just for making other software. In other words, an IDE is a programmer's tool for writing code, organizing project files, debugging applications, and deploying finished products—an all-in-one development solution. Familiar IDEs include Microsoft Visual Studio, Xcode, Eclipse, and countless others.

You'll find mentions of the Eclipse IDE more than a few times in this book in regard to Flash Builder. We'll often refer to Eclipse when we're talking about a feature that isn't specific to Flash Builder but is part of the default Eclipse IDE, which Flash Builder extends.

So what's the deal with Eclipse? By default it comes packaged as a Java editor, but it can handle just about anything you throw at it if you're using the right plug-ins. A lot of Java programmers use Eclipse as their main development environment, but it's also great for JavaScript, HTML, C, Python, and many other languages.

Eclipse also supports source control when you're sharing and maintaining code with a team. And if you're interested

in connecting to a Subversion (SVN) repository, check out Subclipse, a plug-in you can use with Eclipse and Flash Builder. Find it at <http://subclipse.tigris.org/>.

Adobe chose to build Flash Builder on top of Eclipse because of the IDE's popularity with Java programmers, and because the basic package offered a number of mature features. This allowed Adobe to concentrate less on building an entire IDE and more on extending features specific for Flex development. Plus, since Eclipse was already cross-platform, they didn't need to create separate code bases for Mac, Windows, and Linux.

Eclipse is most useful for developments involving multiple languages, because the one IDE can manage everything. More to the point, since Flash Builder uses Eclipse as its foundation, you're free to pick and choose value-adds from the wide assortment of Eclipse plug-ins and use them inside Flash Builder. This opens some more doors, and you'll probably appreciate it sooner or later.

And just in case they ever release Trivial Pursuit, the IT edition, Eclipse was built in Java, and it was inherited from an IBM project, becoming fully open sourced in 2001.

Flash Builder Installation

Once you have a copy of Flash Builder, open the installer and follow the on-screen instructions. It may ask you a few questions about where you'd like to put your installation and things of that nature; unless you're picky, just trust the defaults. When everything's finished, open Flash Builder, and you should be greeted with a screen that looks like Figure 2-5.

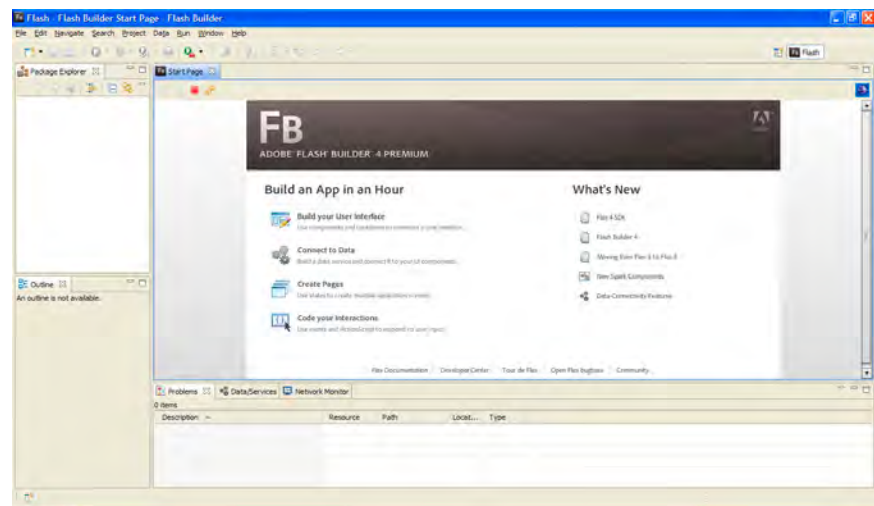


Figure 2-5. The Flash Builder Start Page

This is the Flash Builder Start Page, essentially a browser window running in Eclipse (yes, it has its own browser). The start page offers the “Build an App in an Hour” tutorial for you to get your hands dirty in Flex, and it’s a great introduction. Since you have this book, it’s not necessary, but you might enjoy roaming through some of the materials.

Notice the navigation options at the bottom of the page, specifically those for Flex Documentation, Developer Center, and Tour de Flex (Figure 2-6). If you’re in a hurry to get rolling, you’ll probably blow right past these, but don’t forget to hit them up sometime. Between the Flex Documentation and the Developer Center, there is a ton of information you’ll appreciate as a developer, particularly as the concepts start to crystallize. We know you’re anxious to get started, so let’s take a peek at the Flash Builder tools and jump into our first exercise.

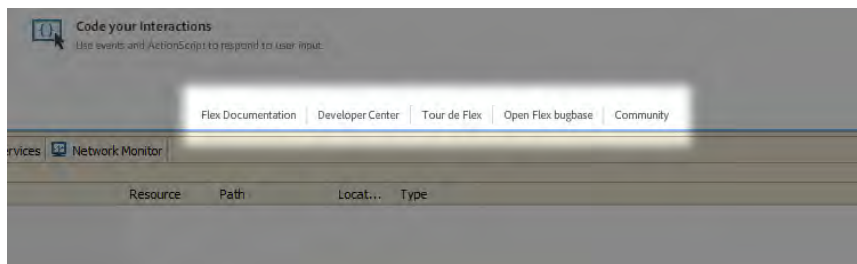


Figure 2-6. Don’t forget to check out the developer documentation from time to time. There’s a wealth of information in those resources.

Your Editor at a Glance

Eclipse and Flash Builder utilize panes and panels to organize the various tools into so-called perspectives aimed at specific development tasks. *Panes* refer to very specific Flash Builder utilities, whereas *panels* are used to dock similar panes. *Perspectives* refer to particular arrangements of panes and panels.

With your fresh install, Flash Builder should open directly into the development perspective. Take a look around. The Package Explorer is where you’ll work with code files, assets (e.g., images, sounds, etc.), and third-party libraries. The Outline pane shows a tree-style visualization of an application’s structure, and the Problems pane will announce any and all complaints—such as bugs and warnings—that the Flex compiler may find in your code. Luckily, because you’re starting fresh, you don’t have any problems yet! We’ll go into more detail about the workbench tools later, but for now let’s have some fun and run an application.

NOTE

The development perspective and the debugging perspective are the perspectives in which you’ll spend most of your time. A perspective is just a particular pane and panel layout arranged to support a specific development activity, such as developing or debugging. In addition to these two preset perspectives, Flash Builder allows you to create and save your own perspectives.

NOTE

You can always add additional Flex applications into an existing project, but a Flex application can't exist outside of a Flex project.

WARNING

Web addresses pointing to resources on the companion website are case-sensitive. Please keep this in mind. We apologize in advance if this creates any confusion.

Running Your First Application

Note that everything is a project in Eclipse/Flash Builder. This means you'll never have just one standalone file; instead, you'll have a group of files that work together to make things happen. So, to create a Flex application, you'll first need a Flex project. In this case, though, we'll start by importing an existing project.

Importing a Project Archive

For the first exercise, we'll import an existing project into Flash Builder.

We've provided a project for this exercise on the book's companion website, so all you need to do is download it. To get started, point your browser to the following URL and save the resulting ZIP file anywhere you please:

<http://learningflex4.com/code/ch2/GettingStarted.zip>

Once you've downloaded the ZIP file, go to Flash Builder and, as shown in Figure 2-7, choose File→Import Flex Project (FXP). The Import Flash Builder Project dialog box should then appear (Figure 2-8). This dialog box can take either a zipped project archive or an unpacked (i.e., a copied and pasted) project folder and import it into the Flash Builder workspace.

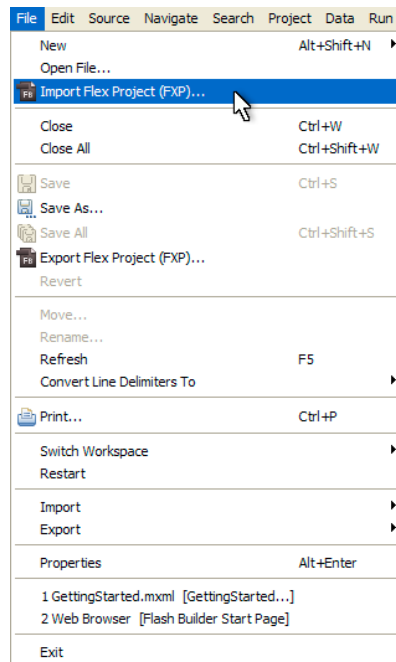


Figure 2-7. Launching the Import dialog

In this case, make sure to choose the File option, because we'll be importing the compressed project directly. Then, click Browse and locate the ZIP file you just downloaded (Figure 2-8). Of course you can import the project anywhere you want, but for this example we chose to import into the suggested default workspace. Notice that the default workspace is nested rather deep in the file structure. We'll spend some more time discussing this in the next chapter, but for now, we want to get you rolling, so just click Finish to finalize the import.

WARNING

If your browser unzips the archive when you download the file or if you've unzipped it yourself, the Import Flex Project Archive command won't work. Instead, you'll want to use the Import Existing Projects into Workspace dialog box, which we'll explain later in the section "Importing an Existing Project."

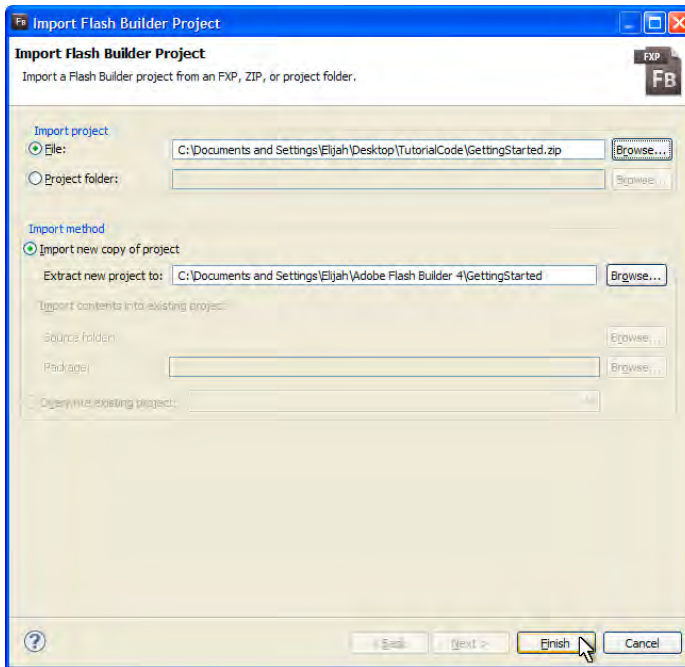


Figure 2-8. Preparing to import a project archive

After a few moments, you'll see your project in the Package Explorer pane. Click the arrow next to the project name in the Package Explorer (Figure 2-9) to expose the contents of the folder. You'll see even more folders, but one of them contains the main application. Which one is it? Well, it's a common practice among programmers to include the source code in a folder called *source* or *src*. So, expand the *src* folder as shown in Figure 2-9, and you'll see the application file *GettingStarted.mxml*.

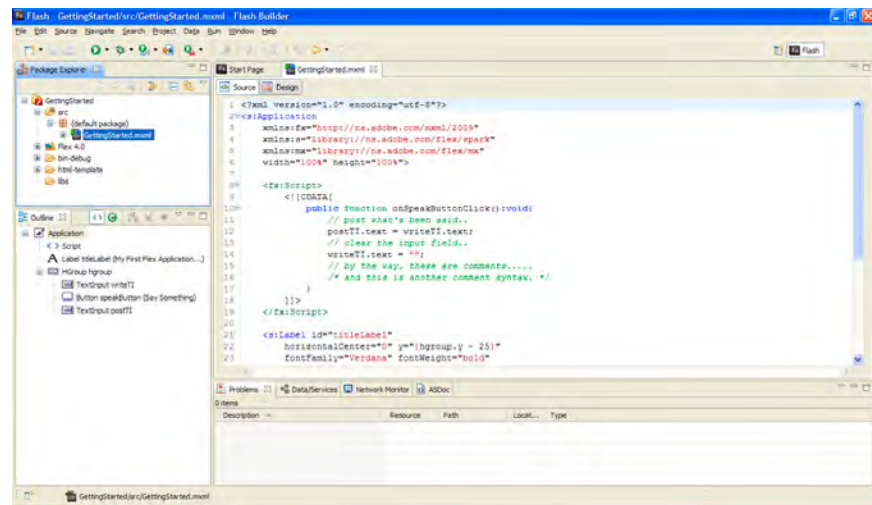


Figure 2-9. Your first project

Opening the Application

You've downloaded a project and imported it into your workspace, and now you're ready to look at the code. Double-click the *GettingStarted.mxml* file, and it'll open in Flash Builder's Source mode.

Flash Builder provides two editor views you can use for development: Source mode and Design mode. Source mode allows you to create and modify the written code of your project. Alternatively, Design mode allows you to visually compose your project by dragging and dropping components, tweaking their styles and arrangements, and previewing the overall application layout as you build it. The buzz phrase for this approach is What You See Is What You Get, or WYSIWYG. Go ahead and jump into Design mode by clicking on the Design button, as shown in Figure 2-10.

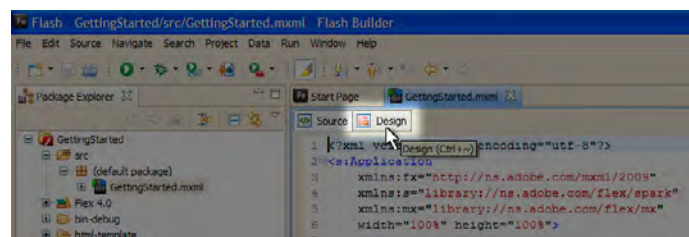


Figure 2-10. Switching between Source mode and Design mode

You might be a little surprised by what you see in Design mode (shown in Figure 2-11). The project does look unsettling but there are perfectly good reasons why. We promise it'll look better when you run it.

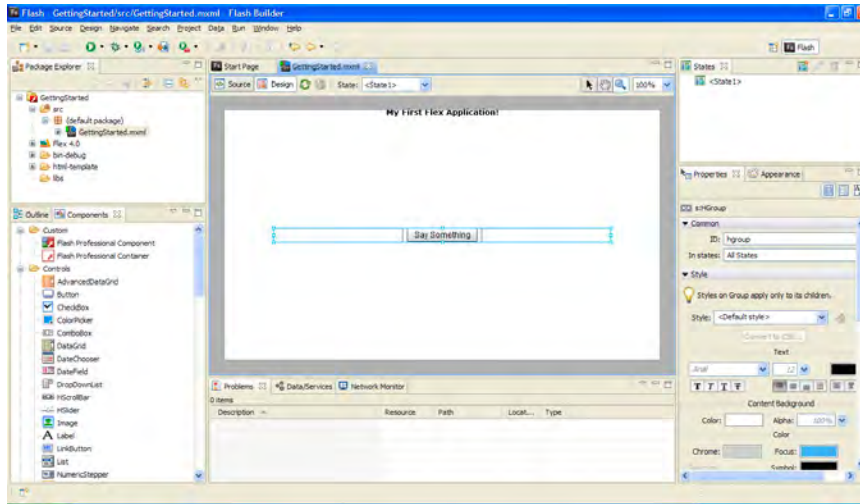


Figure 2-11. The GettingStarted project, a bit half-contrived as of yet

What you read in the next paragraph may not make much sense, but encountering some terms now might prime you to absorb their meaning later when we cover the topics more thoroughly. So, if you can stand a little jargon, here's a description of the layout calamity you're seeing in this first application:

*"The vertical position (**y** attribute) of the **Label**, "My First Flex Application", is calculated at runtime (when the application is launched) by a data binding (a dependent value) to the vertical position of the **HGroup** container that holds the **Button** and the two **TextInput** controls."*

If you're curious about which line of code is handling that so-called "data binding" we just mentioned, jump to Source mode using the keyboard shortcut Ctrl-tilde (~) and look for the statement `y="{hgroup.y - 25}"` in the following code block:

```
<s:Label id="titleLabel"
  horizontalCenter="0" y="{hgroup.y - 25}"
  fontFamily="Verdana" fontWeight="bold"
  text="My First Flex Application!"/>
```

We tried to make it easy to identify the line of code responsible for the data binding. Again, don't worry too much about that previous explanation. We just wanted you to get a quick dose of jargon. This stuff will gradually gel together. For instance, we'll cover data bindings properly in Chapter 8, but for now, let's run this and see what happens.

Running the Application

Now that the *GettingStarted.mxml* file is open in Flash Builder, you can run the application in a browser by selecting Run→Run GettingStarted or by clicking the green, circular “Run” button in your main toolbar, shown in Figure 2-12. The Run button will launch a browser window after a few moments, and you’ll see our first Flex application running in an HTML page (Figure 2-13).

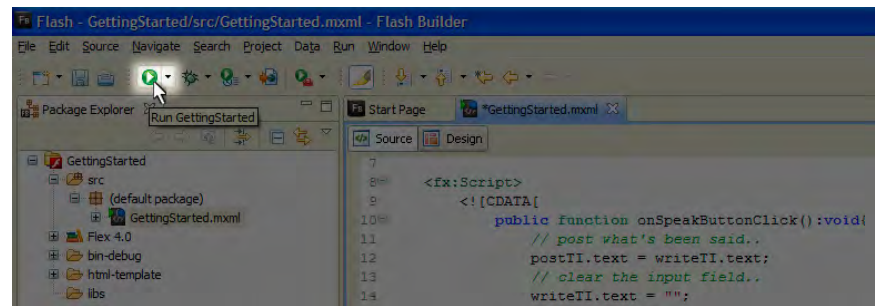


Figure 2-12. The Run button

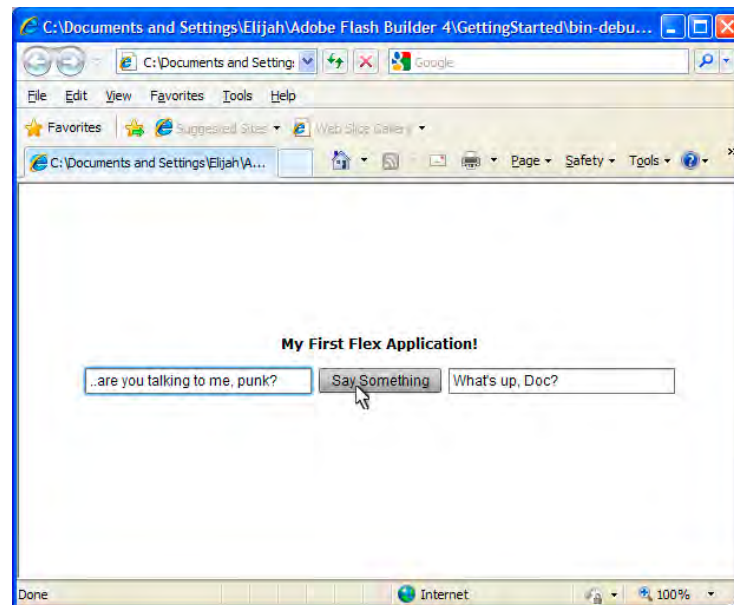


Figure 2-13. The first app, running, and with plenty to say

Everything Is a Project

So now that you've imported your first project and run it, let's delete it. Yes, all that hard work, and you're going to throw it all away! Not to worry, though: you can always bring it back. In Flash Builder, deleting a project from your workspace doesn't mean you have to delete the actual source code, and just because source code exists on your hard drive doesn't mean it's available to Flash Builder. Let's discuss how this works; just follow along.

First, select the project in the Package Explorer—that is, select the top folder titled *GettingStarted*, and don't worry about the files or folders under it. Next, choose Edit→Delete. You'll be prompted with a dialog box asking whether you'd like to delete the contents of the project (Figure 2-14). Be sure to select "Do not delete contents" (which should already be selected) because this will keep the project files on your machine.

NOTE

We'll frequently tell you how to access commands using the menu bar, but many of the most common commands are available as context menus in Flash Builder. For example, to delete the project, you could also right-click the project folder (Control-click on Mac OS) and choose Delete from the context menu.

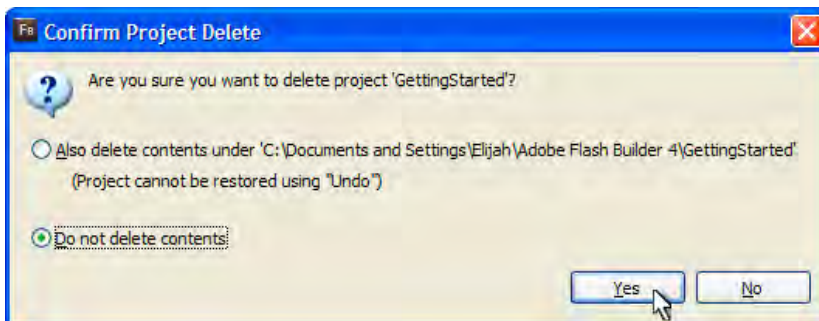


Figure 2-14. Deleting a project

Now you can check your filesystem for the project folder. By default it's under your documents folder and Flash Builder 4, or wherever you originally placed the project. The project should still be there, and you should see a directory structure similar to what was displayed in the Package Explorer.

So, what's the point? Well, it shows you that a project in Flash Builder maintains its own identity. Just because you have some Flex code lying around doesn't mean Flash Builder can use it—not until you reference it as a project. You can try this by going to the *src* folder and double-clicking the *GettingStarted.mxml* file on your machine. It might open in a text editor, or your computer may not even know what to do with it. You might expect it to open in Flash Builder, but that's just not the case. If you want to use it, you have to return to the Import dialog and reference the containing project folder so that Flash Builder can understand how to handle it.

Let's do that, actually, and import the project again.

The Flash Builder Workspace

We'll occasionally refer to workspaces in Flash Builder. A **workspace** is any folder on your system that you choose for saving projects and preference settings. You can have more than one workspace, but the one you create when you install Flex will become your main workspace, and by default it's located under your documents folder and Flash Builder 4. If you want to create a separate workspace, which is reasonable when you have multiple unrelated projects, you can select File→Switch Workspace→Other and create a new one.

Here's a cool tip concerning workspaces: you can copy an entire workspace from one system running Flash Builder and paste it into another system running Flash Builder, and doing so will transfer not only source code from the original system, but editor settings as well (i.e., custom syntax colors, the arrangement of your panes and panels, and any custom perspectives you may have created). In fact, we do this sort of thing frequently when situations require us to work from home or while on the road. This trick can also be really handy if you're a student and you want to keep a workspace intact while shuffling back and forth between a lab computer and your home installation of Flash Builder (the Educational License, of course).

Importing an Existing Project

Importing projects is one of those tasks you'll be doing frequently over time, so it's best to become comfortable with the routine. You might think you already know how to import a project. Previously, you imported a zipped project archive, which is a project in a compressed state. To import an unpacked project—such as a project you previously deleted from your workspace but never archived, or a copied and pasted project—you'll use a slightly different approach.

In Flash Builder, select File→Import Flex Project (FXP) just as before. However, this time choose the Project folder option from the “Import project from” section. This will allow you to import an existing, uncompressed project.

Click Browse, and choose the *GettingStarted* folder (Figure 2-15) you deleted in the previous section. Now click Finish (Figure 2-16), and Flash Builder will import the project.

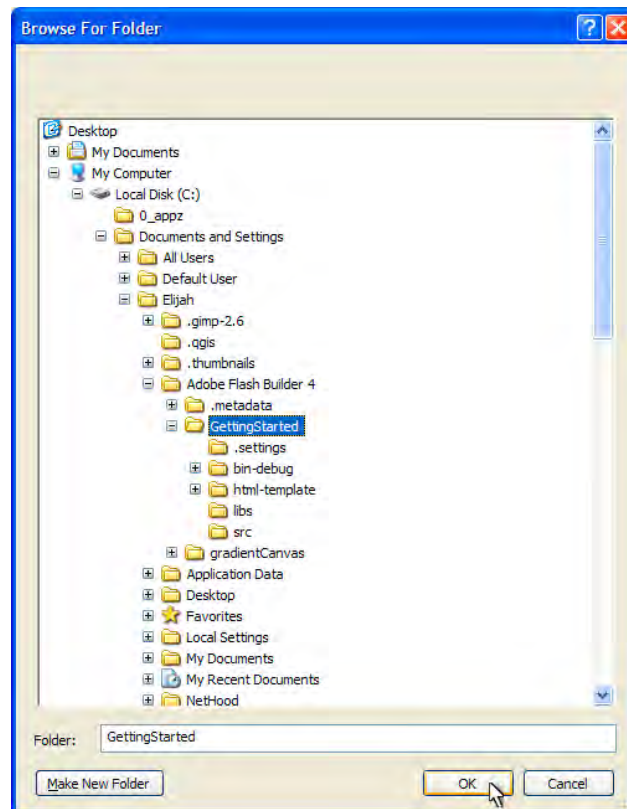


Figure 2-15. Finding the *GettingStarted* project in the default workspace

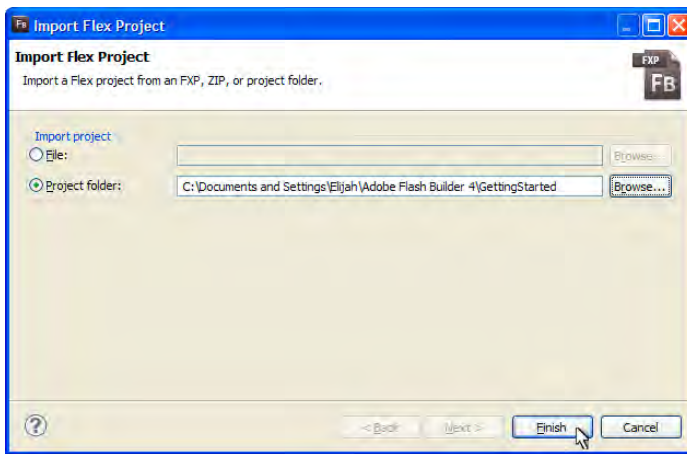


Figure 2-16. Importing a local, unpacked project

Creating a New Flex Project

Of course, when you're creating real Flex applications, you'll usually be creating new projects from scratch. As such, we'll end this chapter by creating a new project that we'll return to in the next chapter.

To create a new project, go to the File menu, and choose New→Flex Project. A dialog box will pop up, giving you options for your project. Give the project a name in the "Project name" field. In this case, call the project *MyNewProject*.

This dialog box has three sections: Project location, Application type, and Server technology. "Project location" lets you modify where the project folder and files will be stored in the local filesystem. The "Project location" value will default to your workspace folder, but you can change it by deselecting "Use default location" and entering your own. For this exercise, just keep the default workspace.

The second section is "Application type." This is where you get to choose between developing for the Web and developing for the desktop with Adobe AIR. In this example, you want your application to run in a web browser, so leave the default type, which is a web application.

Adobe AIR

Adobe AIR is getting a lot of recognition lately, and it's for good reason. With AIR, you can develop desktop applications that provide a Start menu/Dock icon, access to the local filesystem, drag-and-drop support, and several other features you'd expect in a native application. While Flex is useful for creating programs that run in a browser, it's also great for creating media-heavy desktop applications.

WARNING

You can't use spaces or funky characters in your project name—only letters, numbers, the dollar sign (\$) character, and the underscore (_) character are allowed. Eclipse is very good about warning you, though, so rest assured you won't get away with anything.

NOTE

Each Flex project has its own folder, and by default, the project folder typically has the same name as the main application file, but this isn't essential. If you want a new project to be in a different folder, that's fine; if you want it outside your workspace, such as in a folder on your desktop, that's cool, too.

NOTE

You can always change project settings later. This dialog box is just a tool to help you get started. If you ever want to change something, simply select your project in the Package Explorer, and then open Flash Builder's Project menu and choose Properties. Alternatively, you can right-click (Windows) or command-click (Mac OS) a project in the Package Explorer and choose Properties from the context menu.

The third section, “Server technology,” refers to server technologies you might want to use. This setting helps you connect your Flex application to a server environment, such as a development server or a production server. For this simple example, just select None as the server type, because we’re not ready to set up a remote data service. Later, if you want Flash Builder to recognize an external server environment that will support your Flex application, this is the place to start.

You’ll notice the Next, Cancel, and Finish buttons at the bottom of the dialog box. We’re in the mode of keeping it simple, so just click Finish to accept the remaining defaults for your new project.

The Structure of a Flex Project

Now that you’ve created a few projects, it’s a good time to go over what all these folders mean. Every Flex project has a *bin* folder, an *html-template* folder, a *libs* folder, and a *src* folder. Table 2-1 describes the purpose of each of these folders, and Figure 2-17 shows the project folder structure in the Package Explorer pane.

Table 2-1. The Flex project folder structure

Name	Short for	Purpose
<i>bin-debug</i>	binary debugging	Holds the compiled code (the SWF file). For Flex web applications, this also contains an HTML container and supplemental files. For AIR applications, it holds the application descriptor file as well.
<i>html-template</i>	HTML template	Holds the template HTML file that generates the container HTML file (web applications only).
<i>libs</i>	libraries	Holds files for additional compiled libraries.
<i>src</i>	source	Holds the source code. This can be <i>.mxm</i> files, <i>.as</i> files, and an application descriptor file (AIR only).

NOTE

It’s not necessary to preserve these folder names. If you’d prefer to call the source folder *source* instead of *src*, or if you’d like to omit it altogether and keep your main MXML file in the top level of the project, that’s fine. You can arrange this when creating a new project by clicking Next instead of Finish in the New Project dialog box, or you can change the settings later by right-clicking the project and choosing Properties. In the project properties dialog, modify your default folders by going to the Flex Build Path section and changing the Main source folder and the Output folder.

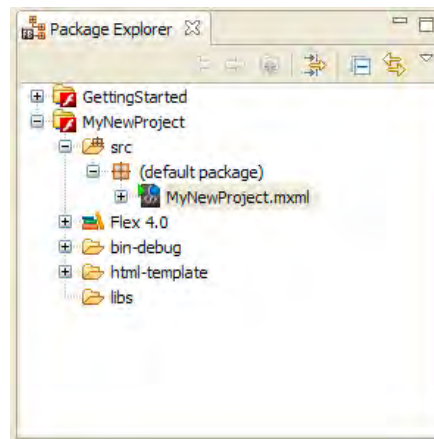


Figure 2-17. Your new project in the Package Explorer

Summary

In this chapter you learned the basics of managing projects in Flash Builder. You also learned how to import, open, and run existing applications, and we discussed the folder structure of a Flex project.

Now that we've covered some Flash Builder essentials, you're ready to have some fun and build something. In Chapter 3, we expand on what you just learned by creating a simple layout in Design mode.

Oh, and as for the last project you created, go ahead and delete it. You can delete the code, too.

USING DESIGN MODE

*“Deal with the difficult while it is yet easy;
deal with the great while it is yet small.”*

—Lao Tzu, *Tao Te Ching*

Design mode is Flash Builder’s What You See Is What You Get (WYSIWYG) editor, and it can be a good place to start a new project. Design mode provides tools to visually compose an application layout, which you do by dragging and dropping components directly into the application. Design mode also provides utilities for assigning color and style properties to components; this is particularly helpful for thematic color selection, as you can see changes taking shape.

This chapter introduces you to Design mode by touring the Flash Builder workbench and reviewing some common components and their properties. In the process, we’ll also assemble a Flex application layout.

A Blank Slate: Your First Project

If you don’t already have Flash Builder open, go ahead and launch it. Inside Flash Builder, open the File menu and choose New→Flex Project (Figure 3-1).

IN THIS CHAPTER

A Blank Slate: Your First Project

Adding Components to the Application

Exploring Common Components

Modifying Properties Directly

Summary

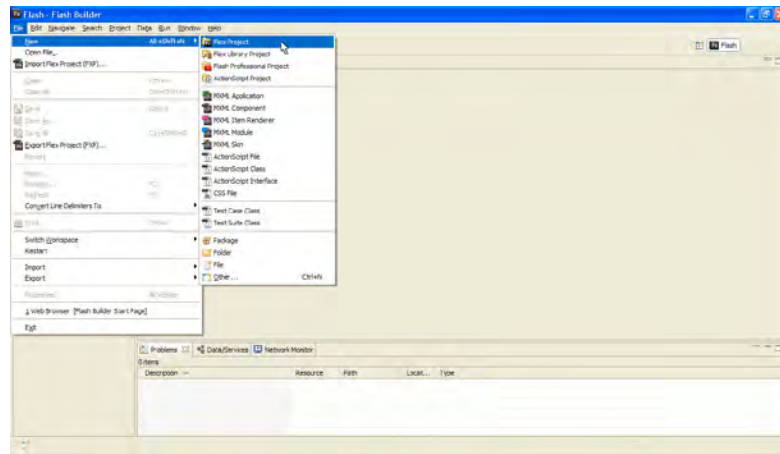


Figure 3-1. Starting a project

NOTE

The default application file created by Flash Builder will inherit its filename from your project name. If this could be inappropriate—for example, if your project has multiple interfaces, such as a public-facing frontend and an administrative backend—you can change the application filename later by right-clicking it (Windows) or control-clicking it (Mac) and selecting “Rename.”

When the New Flex Project dialog appears, enter “HelloWorld” for the Project Name.

Now set the project location. Flash Builder will suggest your default workspace, which is rather deep in your folder structure. For example, the default path for Windows is under the current profile's *Documents and Settings* folder. As such, you may wish to change the project location to something closer to the drive root. You're welcome to keep the default setting, but we'll often use the workspace `C:\Flex4Code`. Below the workspace, each project will exist in a unique directory, so in this case we'll use the project location `C:\Flex4Code\HelloWorld`.

If you endorse this arrangement, your New Project dialog should look like Figure 3-2.

We'll accept default values for the rest of our configuration, so choose Finish to complete the setup. Flash Builder will take a moment to construct the guts of the project, and when it's finished, you'll emerge in Source mode. We cover Source mode in the next section, so let's jump over to Design mode.

Two specific buttons toggle between Source and Design modes, and they are located side-by-side, directly above the code editor. The code editor is the dominant, central window, titled `HelloWorld.mxml`. Click on the Design button to change views (Figure 3-3).

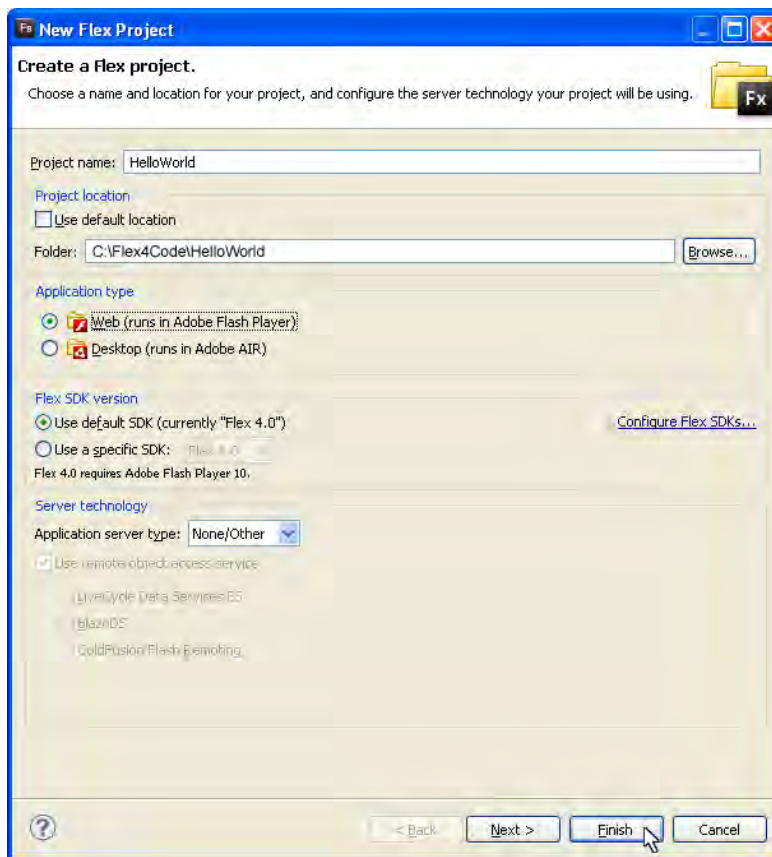


Figure 3-2. Configuring the project

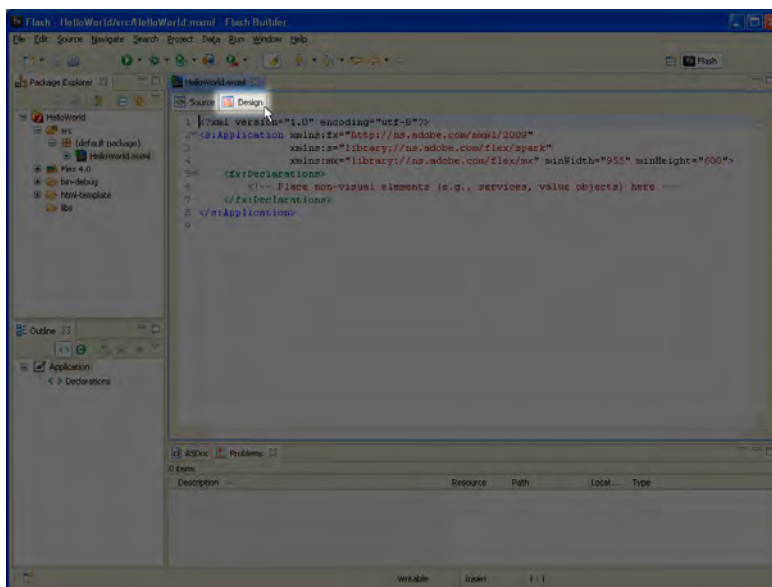


Figure 3-3. Triggering Design mode

NOTE

Flash Builder is built on top of the popular Eclipse IDE. If you've used Eclipse before, you'll be comfortable using Flash Builder. Another benefit of the Flash Builder–Eclipse relationship is the potential to weave Eclipse plug-ins into Flash Builder. Check out one called Web Tools Platform at <http://download.eclipse.org/webtools/downloads/>.

NOTE

If you have multiple monitors, take advantage of them by undocking the perimeter panes and dragging them to a different monitor. This will increase the screen area available to your editor. Undock panes by dragging and dropping them wherever you want. Alternatively, right-click a pane, select Detached, and then drag it elsewhere.

WARNING

If you want to restore your panes and panels to their original positions, open the Window menu and select Perspective→Reset Perspective. This can also be helpful if your multimonitor system is hiding a pane you believe to be visible outside of your screen area, which can happen unintentionally as a result of changing the primary monitor.

Behold Design mode, and your blank slate. We'll add some components momentarily, but first let's glance clockwise around the Flash Builder workbench.

The Flash Builder workbench consists of several toolkits scattered around the perimeter of the application. These toolkits, called *panes*, are grouped together with other panes having a similar theme or purpose—altogether, we call them *panels*. By default, the panels are docked to the edges of the Flash Builder application window.

Front and center is your editor, titled HelloWorld.mxml. The Editor pane will always reveal the filename of whatever file you have open. If more than one file is open, multiple tabs will appear across the top of the Editor pane.

The right panel contains the States pane, and below it, the Properties and Appearance panes. The bottom panel includes the Problems, Data/Services, and Network Monitor panes. Finally, the left panel contains the Package Explorer, Outline, and Components panes. See Figure 3-4.

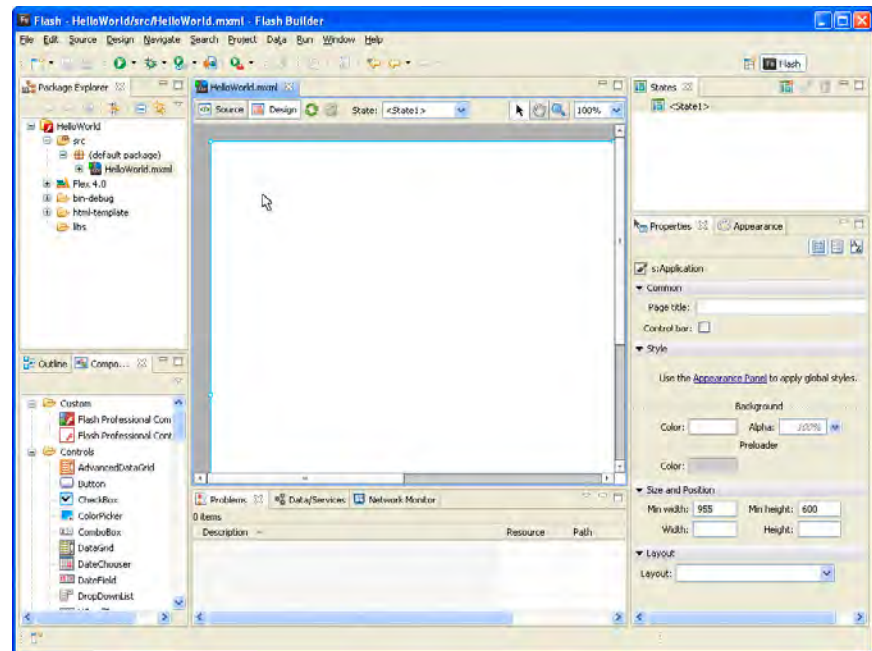


Figure 3-4. The Flash Builder workbench in Design mode

We'll explore the Components, Properties, and Outline panes in this chapter. Because the other panes are consuming valuable screen real estate, go ahead and close them.

Start with the States pane. Close it by left-clicking the “x” in the pane’s title bar. Also close the entire bottom panel (Problems, Data/Services, and Network Monitor). If you prefer, you can minimize the panel, but we won’t be using it.

If you want to restore a pane you’ve closed, you can recover it at any time by opening the Window menu and selecting the pane by name.

Let’s shift our attention back to that blank slate. Presently, the sole representative of your first project is a lonely **Application** container (Figure 3-5). The **Application** container should be selected by default, but if it’s not, left-click anywhere near the middle of your Editor pane to select it. You can tell the **Application** container is selected when it’s highlighted by a light-blue border with “handles” in the corners and along the perimeter.

WARNING

Flash Builder remembers your custom arrangements for each perspective (we briefly discussed perspectives in Chapter 2). At times, you will notice differences between our screenshots and the arrangement on your system. This is certainly the case if you undock your panes. On the whole, however, we’ll try to maintain a default appearance.

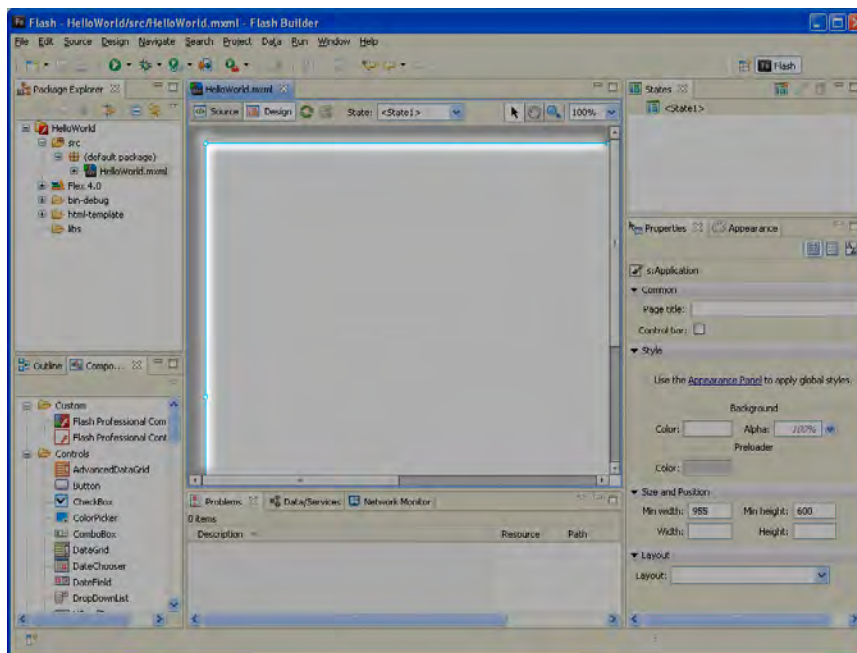


Figure 3-5. The selected **Application** container

We need to tweak a few characteristics of our **Application** container before we add any controls to it, so find your Properties pane, which should be docked on the right. Locate the **Width** and **Height** property fields and change both values to 100%. Clear the values for **Min width** and **Min height**. This should yield a nice, form-fitting **Application** container for your Flash Builder editor (Figure 3-6).

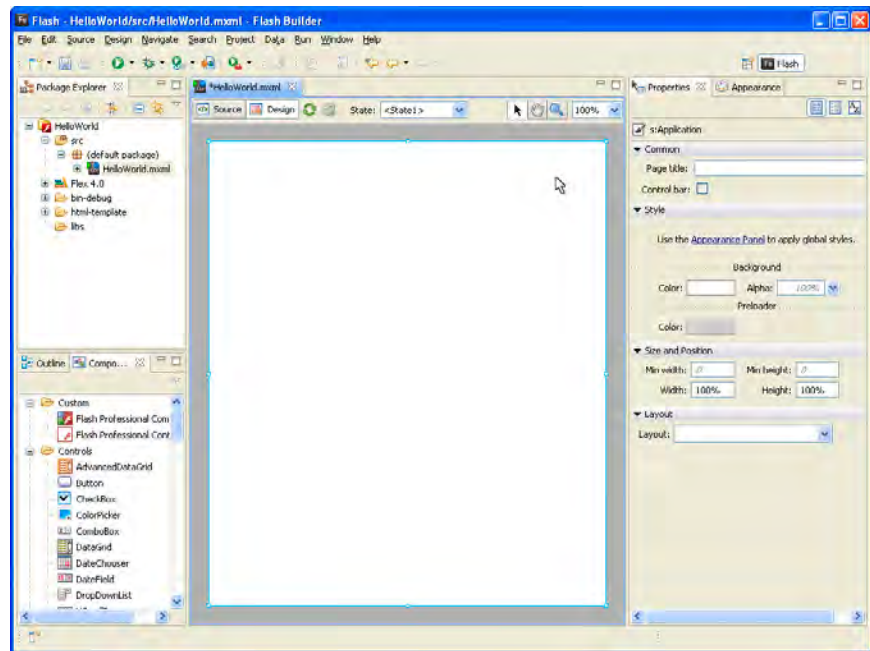


Figure 3-6. The Application container with percentage-based sizes

Now we're ready to add some components.

Adding Components to the Application

As mentioned previously, Design mode lets us add Flex UI components to a project by dragging and dropping. You'll find the common controls and containers in the Components pane.

Dragging and Dropping Components

In the Components pane, scroll down until you notice the Label control, and then drag it into your application and drop it somewhere near the top-left corner. Once dropped, the control should remain selected. Remember, the blue border and the presence of handles indicate that a component is selected. With the label selected, return to the Properties pane to make some changes.

The Properties pane offers access to the properties for the currently selected component, and it has three different views: Standard, Category, and Alphabetical. The Properties pane should be in Standard view by default (Figure 3-7). Standard view is good for quick designing, and it's subdivided into three more categories: Common, Style, and Size and Position.

Under the Common theme, set the label's **id** to `titleLabel`, and set its **Text** property to "My First Flex 4 Application".

Below the Style heading, you can modify typography characteristics such as font family, size, styling, alignment, and color. Let's change some of these. We set the font family to Arial, kept the size at 12, and changed the style to bold. Feel free to take liberties.

But something's vexing us. In the Size and Position area, the **X** and **Y** properties were casually set by dragging and dropping the label control into the **Application** container. If you like to see agreeable, symmetric numbers in situations like these, change the **X** and **Y** values to something like **50** and **50**.

Let's keep adding components.

Scroll down in the Components pane until you find the Layout controls, and drag an **HGroup** container into your editor (don't drop it just yet). We want to drop the **HGroup** near to and below the label.

As you approach the label, notice that snapping indicators will attempt to influence your drop (Figure 3-8). Maybe this is good and maybe it isn't. If you're irritated by the snapping cues, they're easily disabled by holding the **Alt** key while you drag the component through the layout. Alternatively, you can permanently disable snapping by opening the **Design** menu and unchecking the option **Enable Snapping**. In keeping with the old adage "build it fast and fix it later," we recommend using snapping while in **Design** mode, and then fine-tuning component positioning later in **Source** mode.

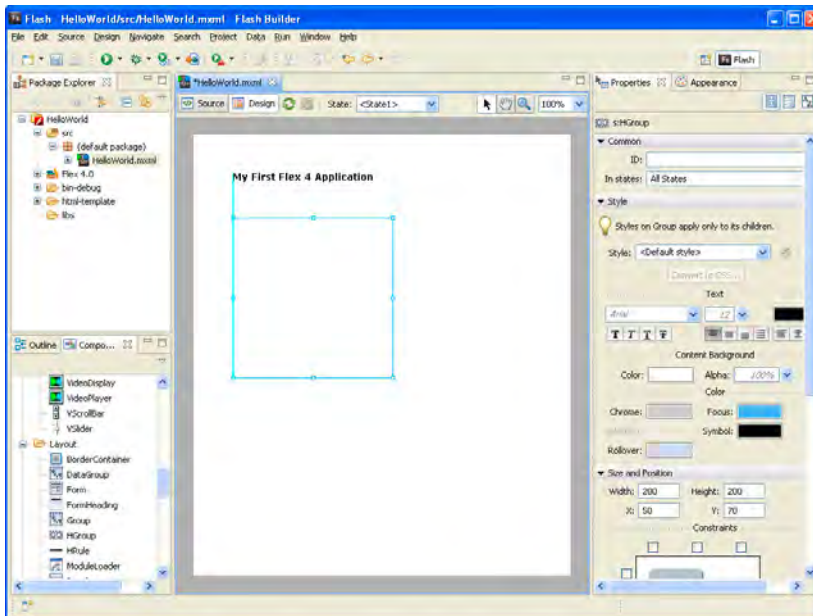


Figure 3-8. Snapping the HGroup container to the label component

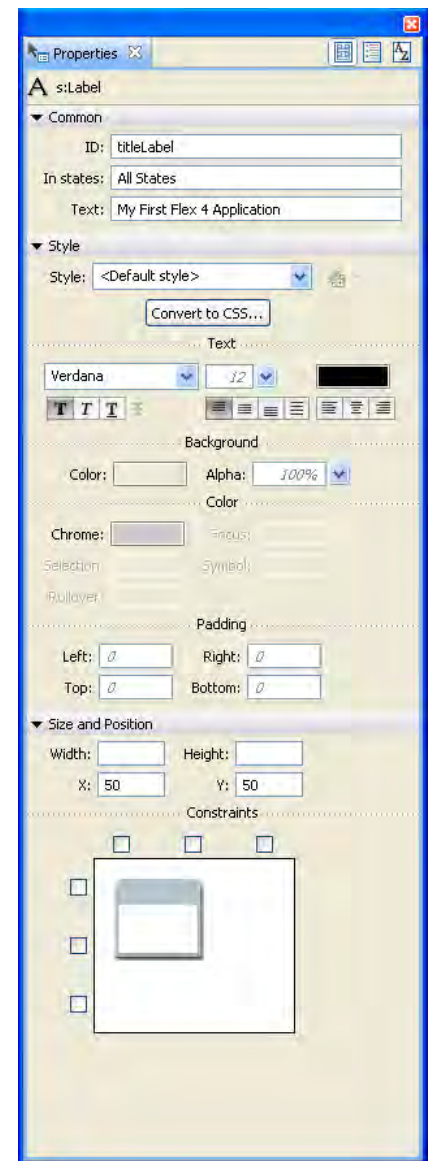


Figure 3-7. A detached Properties pane in Standard view

NOTE

You can temporarily disable snapping by holding the **Alt** key while dragging components around the layout in **Design** mode.

For a permanent solution, disable snapping by opening the **Design** menu and unchecking the option **Enable Snapping**.

Don't Put a Label on That!

Several controls use a **label** *property* to identify their purpose or function. The **Button** component uses a **label** property, and so do **CheckBox**, **RadioButton**, and **Panel**.

However, the **Label** *control* uses a **text** property in the same manner, and it's limited to a single line of text. Therefore, if your UI requires a multiline description, consider using a **Text**, **TextInput**, or **TextArea** control rather than several labels.

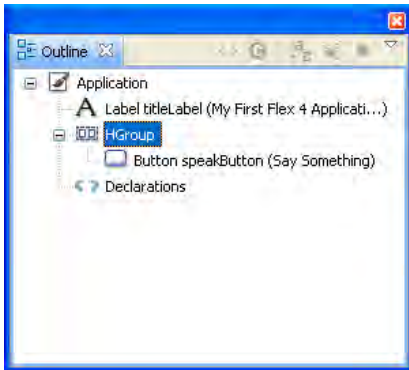


Figure 3-9. A detached Outline pane and selected HGroup container

Fathers and Sons

In the Flex/Flash (and MXML/ActionScript) parlance, any component referred to as a child is implied to be contained by another component. In this example, our **Button** is a child of the **HGroup**. However, if we nest our **HGroup** within a different component—such as a **VGroup**—the **HGroup** would be a child with children of its own.

When you find a suitable place for your **HGroup** container, drop it and change its **width** to 80%. Keep the default **height** value; we'll return to it in a moment.

Next, drag a **Button** control into your **HGroup** container and drop it. Flash Builder will force the button into the top-left corner of your **HGroup**. That's fine. While the button remains selected, set its **id** property to **speakButton**, and change its **label** property to "Say Something".

We can now return to our **HGroup** to fix its **height** property, but we need to select it before the Properties pane will recognize its attributes.

Selecting Components in the Outline Pane

When you want to select a component in Flash Builder, there are two approaches. In Design mode, you can always left-click on a visible component to select it. However, if a component is out of sight (either invisible or masked by a component in front of it) or if many components are cluttered together, the Outline pane offers a more elegant selection mechanism. Let's demonstrate.

Unless you've moved it or closed it, you should find the Outline pane docked on the left, and it should be in the same panel as your Components pane. If you closed it, you can get it back by opening the Window menu and selecting Outline.

Within the Outline pane, left-click on your **HGroup** container. This should select the **HGroup** in your Editor pane (Figure 3-9). You can now tweak its attributes in the Properties pane. Go ahead and delete the **height** value, and then watch the **HGroup** control conform to the height of its child component.

It may be difficult to perceive the real benefit of the Outline pane in the context of this example. However, large applications requiring a plethora of components are much more difficult to navigate by clicking around in the editor. If you relentlessly pursue the clicking-around approach, sooner or later you'll unintentionally resize elements or drag them ever-so-slightly out of place. So don't forget the Outline pane; it's there when you need it.

We're almost done with our first user interface. Only one more component remains to complete it: a **TextInput** control. Return to your Components pane, find the **TextInput** control, and drag it into the **HGroup**. It should snap directly to the right of the button. While it's still selected, set its **id** property to **speakTI** and the **width** property to 100%. Notice how the **TextInput** control adjusts to fill its **HGroup** container.

That's it! You just completed your first user interface, as shown in Figure 3-10. In the next chapter, which discusses Source mode, we'll add code to make this application interactive. For now, let's break to discuss the common components and their properties.

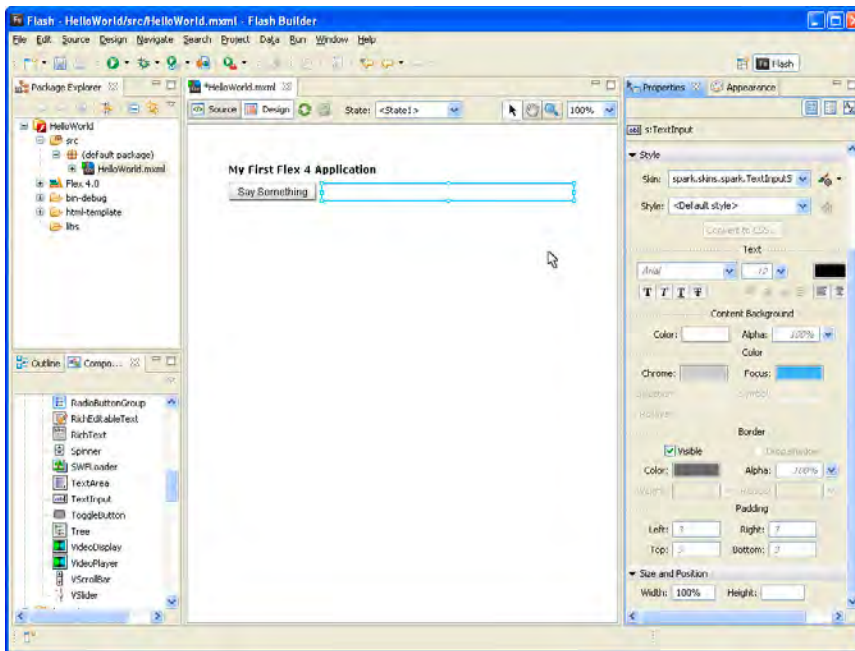


Figure 3-10. The completed HelloWorld user interface

Exploring Common Components

A component is simply a reusable framework object that can be dropped into your application. The very term “component” implies a smaller part of a larger system. The components available to you in Flex come preloaded with helpful properties and methods, and what’s more, they’re easy to style.

Classes, Objects, Packages, and Namespaces

At some point we’re likely to refer to components, controls, containers, navigators, and nonvisual components collectively as **classes** and/or **objects**. These terms come from the object-oriented programming parlance, and they refer to simplified, compact code elements created for specific purposes. In this context, Flex UI components were created to handle communication between the software and a user.

A **package** can be as simple as a few classes (MXML or ActionScript files) saved together in the same folder. However, a package may also be as complex as a folder containing many packages, which may contain packages of their own.

Finally, a **namespace** is merely a unique pointer referring to a specific package. Flex 4 has three native namespaces: Spark (**s:**), Halo (**mx:**), and the Language (**fx:**) namespace.

We’ll discuss namespaces more thoroughly in Chapter 4, and object-oriented programming concepts in Chapter 5.

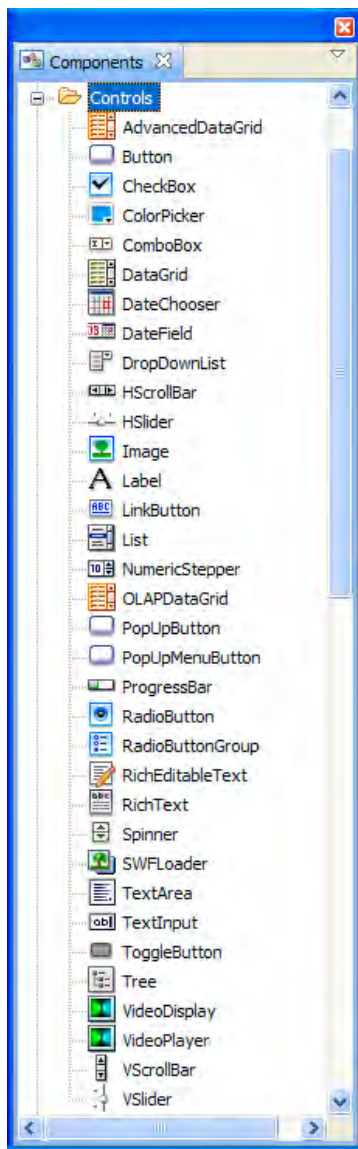


Figure 3-11. Flex controls in the Components pane

Most components fall into one of four categories, and you've already worked with two of them: Controls and Containers. Navigators and Charts are the other two.

Components can be separated further based on their belonging to a package. For example, older Flex components are called *Halo* or *mx* components, and new Flex 4 components are called *Spark* components.

What follows is a brief summary of the common components you're likely to use. We'll discuss the controls first, then containers and navigators.

Controls

Controls are interactive components that typically perform some function—a **Button** clicks, a **ComboBox** presents a menu, a **CheckBox** can be selected and deselected, etc. (Figure 3-11). The **HelloWorld** application required three controls: a **Label**, a **Button**, and a **TextInput**.

Here are the common controls you're likely to use:

Button

This is pretty simple: the **Button** is a clickable control that looks and acts like a physical button.

CheckBox

The **CheckBox** control is much like a **Button**, only it can be toggled to retain a selected or deselected state.

ComboBox

The **ComboBox** provides a compact drop-down menu of options for selection. The **selectedItem** (object) or **selectedIndex** (integer) properties let you determine the current selection.

Image

The **Image** component lets you load external images. Compatible data types include GIF, JPEG, PNG, and other SWFs. Use the **source** property to set a reference to an image.

Label

The **Label** is a simple single-line identifier.

List

The **List** component literally displays a list of items. It allows for item selection and provides scroll bars if there are too many items for the component's allotted size.

ProgressBar

The **ProgressBar** is designed to show download progress. Pair it with an **Image** component by binding its **source** property to an **Image** control's **id** property, and you'll see the download progress.



RadioButton

The **RadioButton** is most useful when grouped with other **RadioButton** controls, so that selecting one **RadioButton** deselects the others. The easiest way to arrange this control is by dragging a **RadioButtonGroup** into a project, which opens a dialog box that helps you create the set of **RadioButton** controls.



Text

Use the **Text** control when you want to display large chunks of text but don't want to have scroll bars. This control will set its size to accommodate its **text** property.



TextArea

Use a **TextArea** when you want text that may exceed the available space. It will add scroll bars as needed.



TextInput

Use the **TextInput** control to collect single-line input. With a **TextInput**, you can get or set its **text** property (this applies to all text components, such as **Text**, **TextArea**, and **RichTextEditor**).

Layout Containers

Layout containers, or simply *containers*, are visual components you use to arrange your UI. Specifically, they organize and align controls in relationship to one another.

Layout containers may also be used to arrange other containers. For instance, it isn't uncommon to see a **VGroup** with several nested **HGroups**, or vice versa.

Application

You don't even have to think about this one. All Flex applications have an instance of the **Application** component as their root. It's a type of container with special constructs that make it perform as the base for everything else. By default, it lets you place components anywhere you like, but you can also make it arrange your components vertically or horizontally.



Group

The **Group** container is one of the lowest-level containers in Flex. It's stripped down and optimized for performance. The **Group** class lends itself to leaner SWFs (leaner = smaller file sizes); however, it accomplishes this by compromising styling potential—it's not skinnable. If you want to skin a **Group**, check out **SkinnableContainer** instead.



BorderContainer

This container supports border and background styling characteristics. If you miss the **Canvas** container, use a **BorderContainer** with a **BasicLayout**.

NOTE

*If you're accustomed to Flex 3, get ready for some surprises, because the Flex 4 Layout containers changed rather dramatically between Halo and Spark. For example, the popular **Canvas** was abandoned, and anything formerly called a **Box** (**HBox/VBox**) is now called a **Group** (**HGroup/VGroup**).*

**Form**

This Halo container makes it easy to create an HTML-like form layout. Pair it with **FormItem** components, and it will stack them vertically with left alignment. The new Spark component set does not include a counterpart for the **Form** class, but the Halo class has a few more miles to go before it sleeps.

**FormItem**

Another Halo holdover, the **FormItem** container takes a control such as a **TextInput** and provides it with a label. You'll set the **label** property on the **FormItem** container itself. When you place a few of these inside a **Form** container, everything will line up automatically, like those HTML forms you're familiar with.

**HGroup**

The **HGroup** organizes its children horizontally. If you want to change the default spacing between components, increase or decrease the **gap** property.

**VGroup**

The **VGroup** is identical to the **HGroup**, except the **VGroup** imposes a vertical alignment.

**Panel**

The **Panel** is a window-like container, complete with a title bar. By default it uses a basic layout, meaning components placed inside it must specify **x** and **y** coordinate positions. You can change the layout to horizontal or vertical as well, which means you can arrange the child controls any way you like.

Navigators

Navigators are a hybrid component designed to handle content visibility. They work much like a layout container and a control mixed together. Navigators take a set of containers and make only one visible at a time, and because of this, they're useful for designing modular interfaces.

Both the Halo and Spark packages offer navigation components, but in Flex 4, you'll find that Halo components still have an important role in content navigation.

**TabNavigator**

This navigator can take a set of Halo containers—such as the **Canvas**, **Panel**, **HBox**, etc.—and make it so only one is visible at a time. This component provides a set of familiar tabs with labels corresponding to nested

containers, and the visible container depends on the selected tab. The **TabNavigator** can take Spark containers such as the **BorderContainer**, **Group**, or **Panel**, but Spark containers need to be nested within a **NavigatorContent** container.



Accordion

The **Accordion** is similar to the **TabNavigator**, except it stacks containers vertically and animates a transition between content selection. The **Accordion** might be used along with a group of **Form** containers, separating them into sections and providing a sense of flow through the data-entering process.



ViewStack

The **ViewStack** is actually an invisible navigator container that's used to provide navigation functionality through other means. Later in this book, we'll demonstrate how the **ViewStack** works as the **dataProvider** for the Spark navigators—**TabBar**, **ButtonBar**, and **LinkBar**.



TabBar

The **TabBar** is a Spark navigator. You'll find it looks quite similar to the **TabNavigator**, only it's a little sleeker. It takes a Halo **ViewStack** with nested Spark **NavigatorContent** containers. The **ViewStack** should be given a unique **id** property and assigned to the **dataProvider** property of the **TabBar**.



ButtonBar

This navigator is assembled just like the **TabBar**, and its functionality is nearly identical. The appearance of the **ButtonBar** resembles several **Button** components placed side by side in an **HGroup** container with a **gap** of 0.



LinkBar

The **LinkBar** is set up exactly like the **TabBar** and the **ButtonBar**; its appearance resembles text hyperlinks separated by pipe (|) characters.



MenuBar

The **MenuBar** is an odd duck. It's a Halo control that operates like a familiar drop-down menu, and its menu selection is handled by an **itemClick** event that the **MenuBar** listens for. Although you'll find the **MenuBar** under the Navigators category in your Components pane, this control doesn't take containers as nested content; rather, you'll need to feed it an appropriate **dataProvider**. XML data is often used as the **dataProvider** for the **MenuBar** control.

NOTE

*One big difference between Halo and Spark containers is this: Halo containers provide a **label** property, and Spark containers do not. A container with a **label** property is necessary for navigators to handle their hal.*

Nonvisual Components

In addition to the components discussed in the previous sections, Flex also provides several nonvisual components to handle functionality that isn't inherently visual. Nonvisual components cannot be dragged from the Components pane and dropped into your application.

Examples of nonvisual components include data objects, formatters, validators, and various service components. For instance, the **HTTPService** class provides methods for connecting to remote data. Similarly, data objects returned by a server, such as an **Array** or an **XMLList**, would also be nonvisual components.

You cannot use Design mode to add or modify nonvisual components, so you need to create them in Source mode, which we discuss in the next chapter. For now, though, let's continue by discussing properties common to visual components.

Modifying Properties Directly

As we learned earlier, you can modify properties of visual components in Design mode via Flash Builder's Properties pane. We changed sizing characteristics, element positioning, element IDs, labels, etc. However, there is more to the Properties pane than the default Standard view.

Examining the Properties Pane

The Properties pane should be docked on the right. If you don't see it, you might need to reenable it by selecting Window→Properties.

As you select components in Design mode, the Properties pane will refresh to show the properties available to the selected component. So, if you select a **Button** component by clicking it or choosing it in the Outline pane, you'll notice the Properties panel refresh to show **s:Button** in its heading area. It will also offer properties relevant to the **Button** class.

Standard view

Standard view is the default view of the Flex Properties panel, and it populates with the most commonly used properties for each component, such as the **label**, **id**, **width**, and **height** attributes. Standard view is also useful for modifying the style characteristics of your components, such as the font, text decoration (bold, italic, underline, and so on), and transparency.

WARNING

Heads up! The Properties pane doesn't appear in Source mode. So, if you're jumping back and forth between Design and Source modes, note that you need to be in Design mode to use the Properties pane.

Category view

Category view is an alternative view of the Properties panel. Category view organizes all a component's properties into a hierarchical arrangement (Figure 3-12). This is similar to Standard view's Common, Layout, and Style themes. However, in Category view, you can look through every property available to a component. It's useful for finding a particular property quickly or for perusing the list of properties available to a selected component.

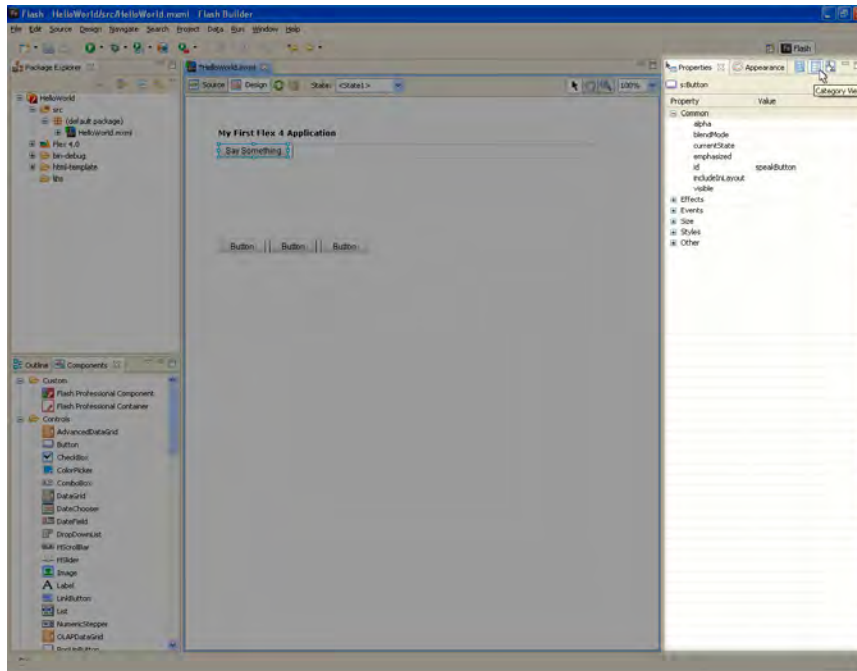


Figure 3-12. Flex Properties pane in Category view

Alphabetical view

The final view available in the Properties pane is the Alphabetical view. This, like Category view, shows all properties available to a component. However, unlike the Category and Standard views, the properties are organized alphabetically, not thematically. It's merely a list of styles and properties organized from A to Z.

Alphabetical view is great when you know the name or the first few letters of the property you want to modify. It's also useful for reviewing which properties have been assigned on a particular component.

NOTE

The Windows version of Flash Builder provides a quick way to get to a property when using Alphabetical view. Click on the Property column heading to select it, then type the first few letters of the property, and the list will scroll to the first match. This technique also works for other panes that have a large list of items. Unfortunately, this functionality doesn't exist for the Mac version.

Common Properties

All Flex visual components inherit from a base class called **UIComponent**. This class has several properties related to visibility, mouse and keyboard interaction, focus, sizing, and all the goodies you'd expect for visual interaction.

Visual components such as buttons, text inputs, and containers *extend* this base component. Component extension refers to components that inherit properties and functions of a less-complicated (i.e., more abstract) component, and that have been expanded to include additional properties or functions geared for a special purpose. This is useful because once you learn the properties of a common component, you can anticipate when it may become useful to extend that component.

What follows is a list of the most commonly used properties for the most common components, along with examples of each:

id

id (short for *identifier*) is a very important property because it's the name you give to a specific *instance* of a component. If you've placed two buttons in an application, calling them "button1" and "button2" allows you to distinguish them from each other later in code. However, it's better practice to use descriptive names based on the component's purpose in your application; for example, "submitButton" and "refreshButton" tell us the real purpose of the component. It's not essential to type an **id** yourself, because Flex assigns a generic **id** as necessary, but it's recommended to do so.

```
<s:Button id="submitButton"/>
```

NOTE

Use of **x** and **y** properties requires a parent container supporting a basic/absolute layout. The **HGroup** and **VGroup** containers will not recognize **x** and **y** property assignments to nested components, because their layout arrangements override explicit positioning (i.e., **x** and **y**) as well as constraint properties (i.e., **top**, **bottom**, **left**, and **right**). We'll provide a better discussion of layout characteristics and component positioning in Chapter 9.

x

This property refers to the number of pixels a component displays to the right of its parent component's left edge. For example, when a **Button** is declared inside a container, and that button's **x** value is set to **20**, the button will display **20** pixels to the right of that container's leftmost edge.

```
<s:Button id="submitButton" x="20"/>
```

y

This property refers to the number of pixels a component should measure down from the top of its parent component. The **y** attribute is similar to **x**, except that it references vertical position instead of horizontal position.

```
<s:Button id="submitButton" x="20" y="20"/>
```

visible

This property controls whether an item is visible in the application layout. Setting **visible** to **false** will make an item disappear from view, but it will still maintain the same space. For instance, if we declare four

buttons inside an **HGroup** container, which aligns them horizontally, and if we make the second button invisible, the two buttons to its right and the one to its left will remain in the same place because the invisible button continues to occupy its position in the layout (Figure 3-13).

```
<s:HGroup>
  <s:Button label="One"/>
  <s:Button label="Two" visible="false"/>
  <s:Button label="Three"/>
  <s:Button label="Four"/>
</s:HGroup>
```



Figure 3-13. Button “Two” is invisible but still occupies layout space

includeInLayout

Setting a component’s **includeInLayout** property to **false** will cause a container to fully ignore that component when rendering its layout (Figure 3-14).

```
<s:HGroup>
  <s:Button label="One"/>
  <s:Button label="Two" visible="false" includeInLayout="false"/>
  <s:Button label="Three"/>
  <s:Button label="Four"/>
</s:HGroup>
```



Figure 3-14. Button “Two” still exists, but it’s invisible and doesn’t take up layout space

toolTip

A **toolTip** is a pop-up message that displays when the mouse cursor hovers over a component for a moment (Figure 3-15). It can help describe an item’s purpose. All Flex UI components support the **toolTip** property, so it’s easy to implement.

```
<s:Button toolTip="Click Me!"/>
```



Figure 3-15. A Button with a tooltip

label

Not all components use labels, but many do. For a **Button**, a **label** is a no-brainer—it’s the text that displays on the button. **CheckBox** components use them similarly (Figure 3-16). Halo containers use labels, but Spark containers do not. For instance, the **FormItem** container will display a label, which can be used as a description of a field when paired with another control, such as a **TextInput**, in a **Form**. For other Halo containers, the **label** displays within navigation components. That is, when using a navigation component such as a **TabNavigator**, a Halo container’s **label** will display as the label of the tab that selects it.

```
<s:CheckBox label="I have read the terms and conditions"/>
<s:Button label="Submit"/>
```

NOTE

There is also a control called **Label**, which is used for single lines of text. The **Label** component doesn’t take a property called **label**; rather, it uses a property called **text**.

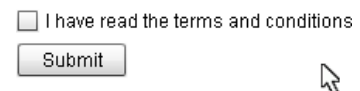


Figure 3-16. A CheckBox and a Button with their own labels

Enter Your Name:

Figure 3-17. Two controls that use the text property

text

The **text** property refers to the text string displayed in text controls such as **Label**, **TextInput**, and **TextArea** (Figure 3-17).

```
<s:Label text="Enter Your Name:"/>
<s:TextInput text="Bingo"/>
```

enabled

Setting an **enabled** property to **false** makes a component unavailable. This causes the component to “gray out” and generally ignore mouse interaction (Figure 3-18). This is useful when you want to disable a control that isn’t ready for use, such as a submit button when required fields are empty or invalid. However, the **enabled** property also affects containers and their children. This means you can set a container’s **enabled** property to **false**, and everything inside will be disabled.

```
<s:HGroup>

  <s:Panel title="Enabled Panel" enabled="true">
    <s:layout>
      <s:VerticalLayout
        gap="4"
        paddingLeft="5" paddingRight="5"
        paddingTop="5" paddingBottom="5"/>
    </s:layout>

    <s:Button label="Enabled Button"/>
    <s:TextInput text="Enabled Text"/>
    <mx:DateField />
  </s:Panel>

  <mx:Spacer height="10"/>

  <s:Panel title="Disabled Panel" enabled="false">
    <s:layout>
      <s:VerticalLayout
        gap="4"
        paddingLeft="5" paddingRight="5"
        paddingTop="5" paddingBottom="5"/>
    </s:layout>

    <s:Button label="Disabled Button"/>
    <s:TextInput text="Disabled Text"/>
    <mx:DateField />
  </s:Panel>

</s:HGroup>
```

NOTE

Curiously, we have found situations where disabled components continue to serve their **tooltip** pop ups.

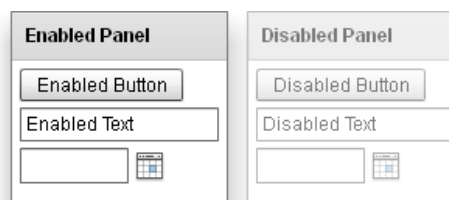


Figure 3-18. Two Panels with the same contents; one is enabled, the other disabled

source

Use the **source** property to point a control, such as an **Image** control, to an external asset. You can also *bind* the **source** property of a **ProgressBar** to the **id** of an **Image** control to measure the control's rendering progress. The following example demonstrates two usages of the **source** property—one for identifying an **Image** control's target, and the other for identifying which component's rendering status a **ProgressBar** should follow (Figure 3-19).

```
<mx:ProgressBar id="progressBar" source="{image}"/>
<mx:Image id="image"
    source="http://www.learningflex4.com/photos/LizzyStudio.jpg"/>
```

LOADING 100%



Figure 3-19. An Image and a ProgressBar working together beautifully

Let's face it, some of these properties will require rote memorization, but when you're using Design mode, the most common options are right at your fingertips. Because many components have commonalities, though, it's not difficult to become familiar with the common properties.

NOTE

*Yet again, we casually mention binding one property value to another. We're referring to data binding, and it's common in Flex. Bound values (and expressions, e.g., 1+2) are wrapped in braces ({ and }), and they indicate dependence on the wrapped value or expression. In this example, **image** is the **id** of the **Image** control, and thus **image** is a variable that can be referenced with binding. We'll provide a serious discussion of data binding in Chapter 8.*

Summary

This chapter introduced you to the common components and their properties. Hopefully you are now comfortable using Design mode to create user interfaces by dragging and dropping components into the editor. You still have a few skills to learn in order to make your applications interactive and fun, but you're on your way to learning Flex. In the next chapter, we'll go behind the scenes with an introduction to Source mode, and we'll continue to work with the **HelloWorld** project, so make sure to save it if you close Flash Builder.

USING SOURCE MODE

“The whole is greater than the sum of its parts.”

—Aristotle

When we created the **HelloWorld** application, Flash Builder automatically generated several lines of MXML code. In this chapter, we use Source mode to examine our **HelloWorld** application and glean some additional insight into the structure of a Flex project. We also use Source mode to add a few more components and get a feel for Flash Builder’s code completion mechanism and MXML syntax.

What Design Mode Does

As previously stated, when you drag components into your Flex 4 project, Flash Builder creates corresponding MXML code in whatever MXML file you’re currently editing. Conversely, if you alter the MXML code in Source mode, Design mode will rebuild and reflect the changes. Let’s take a look at the MXML for our **HelloWorld** application.

If you closed Flash Builder since creating **HelloWorld** in Chapter 3, reopen it and switch to Source mode. It’s easy to switch from Design mode to Source mode; just click the “Source” button above your WYSIWYG editor (Figure 4-1).

IN THIS CHAPTER

What Design Mode Does

Anatomy of a Flex Application

Adding Components in Source Mode

Code Completion

MXML in Depth

S, FX, and MX: Namespaces Explained

Summary

NOTE

Whether you're using Windows or Mac OS, you can quickly switch between Design mode and Source mode using the keyboard shortcut Ctrl-tilde (~).

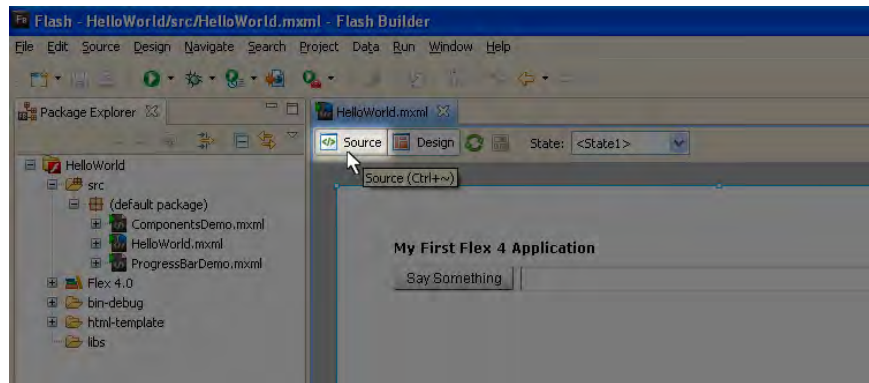


Figure 4-1. Switching to Source mode

Total Recall

Flash Builder remembers the workspace, project, and editors that were open when it was last closed, so it opens where you left it, possibly in **HelloWorld**'s Design mode. If you were browsing around the IDE between chapters (which is fine), it could open elsewhere, and now you know why.

If you deleted **HelloWorld** from your workspace and need to reload it, open the File menu, select "Import Flex Project (FXP)", toggle "Project Folder", and browse to your **HelloWorld** project. We recommended the location *C:\Flex4Code\HelloWorld*, close to your drive root, but if you didn't change this value, the project may be in the default workspace.

Flash Builder recognizes the following default workspaces:

- Windows XP: *C:\Documents and Settings\[user]\My Documents\Flex Builder 4*
- Windows Vista and Windows 7: *C:\Users\[user]\Documents\Flex Builder 4*
- Mac OS: */Users/[user]/Documents/Flex Builder 4*

Anatomy of a Flex Application

Your source code for this example should look something like that in Example 4-1.

NOTE

Code created by Design mode has a tendency to be ugly. On another note, due to limitations of print/page dimensions, it's impossible to identically replicate the code you're seeing in print. As a result, expect our book examples to differ from yours somewhat.

Example 4-1. The MXML code autogenerated by Design mode

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" width="100%"
    height="100%">
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects)
            here -->
    </fx:Declarations>
    <s:Label x="50" y="50" text="My First Flex 4 Application"
        id="titleLabel"/>
</s:Application>
```

```

    fontFamily="Arial" fontWeight="bold"/>
    <s:HGroup x="50" y="70" width="80%">
        <s:Button label="Say Something" id="speakButton"/>
        <s:TextInput id="speakTI" width="100%" />
    </s:HGroup>
</s:Application>

```

Code created by Design mode is usually jumbled and messy. As a result, our source code could use some housecleaning. Before we break apart the code and discuss it, let's clean up the code here in Source mode. Using extra line breaks and tab indents, make your Source code look more like Example 4-2.

Example 4-2. Our clean MXML code after reformatting it in Source mode

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%">

    <s:Label id="titleLabel" x="50" y="50"
        fontFamily="Arial" fontWeight="bold"
        text="My First Flex 4 Application"/>

    <s:HGroup x="50" y="70" width="80%">
        <s:Button id="speakButton" label="Say Something"/>
        <s:TextInput id="speakTI" width="100%" />
    </s:HGroup>

</s:Application>

```

NOTE

Notice that we deleted the **Declarations** block? This block will always appear in a new project, so if you won't need it—and we won't need it for a while—you can delete it to create cleaner code. Of course, it won't hurt to leave it. It's your decision.

Now that we can comfortably read the code, let's consider what's happening. The first line is a mere XML declaration that usually begin XML files. It has no functional purpose in the code, but Flash Builder will always add it to new application files. The “real Flex” begins with the Spark **Application** tag.

Every Flex application contains a root tag. Flex applications geared for web deployment will use an **Application** tag, but Adobe AIR applications designed for the desktop will use a **WindowedApplication** tag. Our **HelloWorld** application is designed for the Web; therefore, the entire project exists between **<s:Application>** and **</s:Application>**, the opening and closing tags of the Flex **Application** container.

Notice that the opening tag includes an ensemble of attributes, specifically several namespace assignments (i.e., **xmlns:**), as well as the **width** and **height** properties we established back in Design mode. The opening tag for every component exists between left (<) and right (>) angle brackets.

The **Declarations** section (which you may have deleted per the note) is identified by the **<fx:Declarations>** and **</fx:Declarations>** tags, and it's reserved for nonvisual elements. We briefly discussed nonvisual elements in Chapter 3. Even though this application doesn't have any nonvisual elements, Flash Builder creates the Declarations section anyway.

The `<s:Label/>`, `<s:HGroup/>`, `<s:Button/>`, and `<s:TextInput/>` tags represent components we added in Design mode. The `Label` and the `HGroup` container are nested within the main application, but the `Button` and the `TextInput` are nested within the `HGroup`.

The `Label`, `Button`, and `TextInput` controls are closed differently than the `Application` and `HGroup` containers. Closing XML tags using `/>` is an optional syntax, but it's frequently used for controls. Alternatively, you could open and close the `Button` using the code shown in Example 4-3.

Example 4-3. A `Button` created with optional syntax

```
<s:Button id="speakButton" label="Say Something"></s:Button>
```

However, using the full syntax on controls has a tendency to make your code less readable. Fortunately, Flash Builder will encourage the abbreviated syntax by using it for controls created in Design mode.

As you inspect the remaining MXML code, you should recognize other properties we set back in Design mode. While you're here, feel free to change some of the basic properties (`x`, `y`, `width`, `height`, `text`), and then leap back into Design mode (Ctrl-tilde) to see the changes take effect.

Adding Components in Source Mode

Of course, we can also add components to our project while in Source mode, so let's add another button to perform a "clearing" function.

Just below the `TextInput` but still within the `HGroup` tag, add a `Button` component by typing `<s:Button id="clearButton" label="Clear"/>`. In context, it should look like Example 4-4.

Example 4-4. Adding a `Button` in Source mode

```
<s:HGroup x="50" y="70" width="80%">
  <s:Button id="speakButton" label="Say Something"/>
  <s:TextInput id="speakTI" width="100%"/>
  <s:Button id="clearButton" label="Clear"/>
</s:HGroup>
```

If you switch back to Design mode, you'll notice the `TextInput` control sandwiched between the "Say Something" and "Clear" buttons. The `HGroup` container is forcing the three controls to consume the 80% width allotted to it.

For the sake of playing around, convert both `HGroup` tags to `VGroup` tags (literally replace the "H" in each tag with a "V"), and then jump back to Design mode to see how this change affected the layout of your application (Figure 4-2). You can keep the layout in this arrangement if you want, but our example will continue using the `HGroup`.

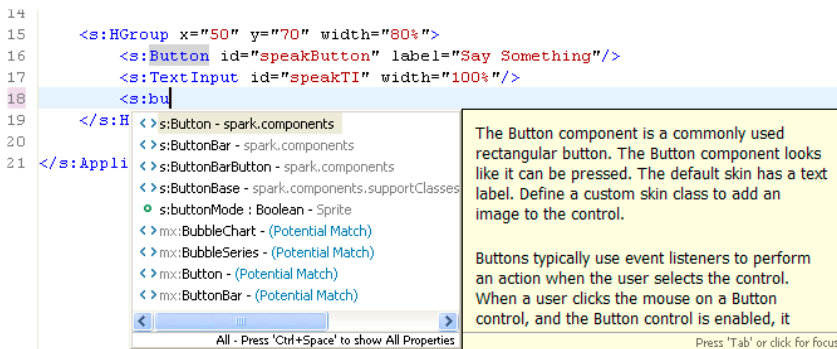
My First Flex 4 Application

My First Flex 4 Application

Figure 4-2. HelloWorld using HGroup and VGroup, respectively

Code Completion

When you created the button in Source mode, did you notice a pop up after you started entering the code for the control? Perhaps it looked like Figure 4-3.

*Figure 4-3. Flash Builder's code completion utility*

Code completion is your best friend when you're new to a programming language. Not only does code completion help you avoid typos and spelling errors, it can also be a useful reference. Consider this: in Design mode's Properties pane, you can toggle Category or Alphabetical view to see properties available to a component. Similarly, Source mode's code completion allows you to browse a control's properties and methods, but it also allows you to browse components, classes, data types, objects, etc. If you're trying to brainstorm a solution to a programming task, sometimes it's helpful to trigger code completion just to peruse the menu. You never know; it just may trigger some inspiration.

Code hinting will attempt to offer you the best reasonable match to the text you've entered into the code editor, and it will begin making suggestions alphabetically as soon as you create the left angle bracket (<).

NOTE

Code completion is a rose with many names: code hinting, content assist, auto-complete, and autocompletion, to name a few. Developers with a lot of experience using Microsoft's Visual Studio might even call it IntelliSense. This is important because you'll encounter these various terms in the blogosphere.

What Gives?!

If you're not receiving code completion assistance as you type, it may indicate a syntax problem somewhere in your code. Basically, Flash Builder's Flex parser got lost, and it's having trouble distinguishing between objects, properties, and their attribute assignments.

So, if you're no longer getting code hints, finish your most recent statement and save the file (in Windows, Ctrl-S; in Mac OS, Command-S or File→Save) to see whether you're alerted to any errors in the Problems pane.

We'll discuss debugging and problem identification later, in Chapter 6.

NOTE

As you become more familiar with Flex components, you may type fast enough that the code hinting doesn't appear. If this happens, or if you did something to lose the utility, use a keyboard shortcut to get it back. In Windows, it's Ctrl-spacebar, and in Mac OS it's Command-Shift-spacebar.

If you're looking for a specific component, type as much of the component declaration as you find convenient (e.g., "<s:tex"); then, once you get the content assist utility, either left-click on the component you want or hit Tab to change focus to the code completion pop up. Once the code hinting utility has focus, use your arrow keys to scroll through the menu. Even better, if you typed enough text for code completion to zero in on the correct component, simply press Tab and then Enter to select it and plug it into your code. That declares your class.

After you've declared a class, use code completion to configure its properties and methods. So, if you just declared a new instance of the **TextInput** using code completion, you'll have **<s:TextInput**, with your insert cursor against the last character. But once you hit your space bar, you'll get a host of properties and methods to choose from.

MXML in Depth

Metaphorically, think of MXML as the skeleton within the body of your source code. Just as skeletons provide structural definition to our human bodies, MXML code gives your project a visible structure.

You'll quickly adapt to MXML syntax because it's so intuitive, but a few details should be covered now to guide your thinking.

The XML in MXML

XML stands for *Extensible Markup Language*. Extensible refers to the capacity to create your own tag relationships. Markup means your tags can include text as well as extra information about that text, also via tags. Remember, tags are identified by their <opening> and </closing> syntax.

XML is *not* a programming language. Rather, XML is a structured arrangement of text and data. There are no keywords, component classes, class properties, or methods. It's simply textual information presented according to specific syntax rules that identify a hierarchy or relationship between the elements. So how do XML and MXML compare?

MXML borrows its syntax rules from XML. However, to the Flex compiler, MXML identifies ready-made ActionScript classes that can be *instanced* using XML syntax. In this way, MXML is essentially a class vocabulary that provides a more convenient approach to declaring ActionScript UI components.

It's All About Structure

As we just discussed, XML is simply structured text. It distinguishes data structures within tags defined by angle brackets (< and >). There isn't an XML vocabulary of reserved words; in fact, the author of XML is free to create her own structural hierarchy. So, XML is purely syntax and structure.

You're reading a book right now, so you recognize that a book is divided into chapters and sections. If you wanted to represent a book in XML, you could create tags like `<book/>`, `<chapter/>`, and `<section/>`, and deploy the book like Example 4-5.

Example 4-5. *Empty XML demonstrating structural hierarchy*

```
<book>
  <chapter>
    <section/>
  </chapter>
</book>
```

As implied by this example XML, “book”—the root node—contains “chapter” nodes, which contain “section” nodes.

A Few Ground Rules

XML parsers, including Flash Builder, web browsers, etc., are *unforgiving* when it comes to XML syntax. While you can invent whatever tags and data structure you desire, you *can't* invent your own syntax rules. So let's discuss the basics of XML syntax.

All that is opened must be closed

An important fact about XML is that each tag must be complete. If a tag is opened, it must eventually be closed later in the XML. As you've seen, an XML tag is opened with a left bracket (`<`), some text representing the tag's name, and then a right bracket (`>`). To maintain our previous example, `<book>` is an opening tag. Finishing the tag requires closing it using a forward slash (`/`) in the proper position within the closing tag, `</book>`.

You can ensure all tags are closed in a couple of ways. The first is by creating the end tag immediately after you create a beginning tag. So, after creating the open container tag `<book>`, go ahead and create the closing tag `</book>`. Alternatively, if a tag will not have any nested tags—as is often the case with simple Flex controls such as the `Button`, `CheckBox`, and others—you can use shorthand syntax by adding a forward slash immediately before the right angle bracket of the opening tag, like this: `<section/>`. So, `<section></section>` is equivalent to `<section/>`.

Case matters

XML is case-sensitive. That is to say, uppercase letters and lowercase letters distinguish different elements. So, `<book>` and `<Book>` aren't the same in XML. That means `<s:Text>` and `<s:text>` are different as well.

NOTE

Encoding in this context refers to your potential to use various characters and symbols in your XML markup. There are different encoding standards available to you, but **utf-8** is the way to go. For more information, see the w3schools discussion of this topic at http://www.w3schools.com/XML/xml_encoding.asp.

WARNING

Make sure you don't have any spaces, empty lines, or other whitespace before the XML declaration in a Flex application. Otherwise, Flex will warn you that whitespace is not allowed before an XML Processing Instruction and your application won't compile!

Declarations are optional, but polite

The first line of an XML document may optionally contain a line declaring it as XML and listing what encoding it uses. The declaration looks like the line in Example 4-6.

Example 4-6. An XML declaration

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

All MXML files created through Flash Builder automatically contain this XML declaration, so you don't need to worry about creating it.

Because MXML is XML, MXML inherits all the XML rules.

The Anatomy of a Tag

An XML tag can contain information in two ways, either by content or by attributes. *Content* is simply the text that exists between the node's opening and closing tags. *Attributes* are value assignments inside the opening tag. Consider the XML in Example 4-7.

Example 4-7. XML with attribute information

```
<book title="Learning Flex 4"
      author1="Alaric Cole" author2="Elijah Robison">

    <chapter title="Using Design Mode"/>
    <chapter title="Using Source Mode"/>
</book>
```

In this example, **<book>** is the root node, but **title**, **author1**, and **author2** are attributes of that book. This code also has two nested tags representing chapters, and each has a title attribute.

Compare the previous code to Example 4-8, which contains the same information using a different structure.

Example 4-8. XML with content information between the tags

```
<book>
    <title>Learning Flex 4</title>
    <author1>Alaric Cole</author1>
    <author2>Elijah Robison</author2>
    <chapter3>
        <title>Using Design Mode</title>
    </chapter3>
    <chapter4>
        <title>Using Source Mode</title>
    </chapter4>
</book>
```

The second example is essentially the same as the first, but the second is more verbose. The first example uses attributes, and the second uses nested tags. So, attributes can be useful as a compact way to represent information, and more compact means more readable. Compare this code to the same example in MXML.

You're probably used to seeing something like this:

```
<s:Label text="Learning Flex 4"/>
```

But did you guess you could do the following?

```
<s:Label>
  <s:text>Learning Flex</s:text>
</s:Label>
```

Those two code snippets create the same product, but one uses an attribute to add the **text** property, and the other uses a nested tag. We recommend using attributes in your MXML for the reasons stated earlier: compactness and readability.

There will be times, however, when you'll need to use nested tags instead of attributes. This is because nested tags allow more complex content than a single line of text or a number. In other words, nested tags are preferable for adding data that can't be represented as simple attributes. Consider the case of the **text** property. If the text were a whole paragraph, it would look strange as an inline property.

Similarly, nested tags are essential when assigning complex data properties. For example, you might use an **ArrayCollection** to supply data to a control, and some controls can display a long list of items. Example 4-9 shows a relevant example.

Example 4-9. An MXML ComboBox with a nested ArrayCollection

```
<s:ComboBox selectedIndex="0">
  <s:ArrayCollection>
    <fx:Object>Alabama, AL</fx:Object>
    <fx:Object>Alaska, AK</fx:Object>
    <fx:Object>Arizona, AZ</fx:Object>
    <fx:Object>Arkansas, AR</fx:Object>
  </s:ArrayCollection>
</s:ComboBox>
```

The same is true for complex components that may require an array of other component objects, such as the **columns** property of a **DataGrid**, which takes an array of **DataGridColumn** components, each with its own properties. You can see an example of this by dragging a **DataGrid** control to your application in Design mode, which gives you the MXML in Example 4-10.

Example 4-10. An MXML DataGrid with nested DataGridColumn components

```

<mx:DataGrid>
  <mx:columns>
    <mx:DataGridColumn headerText="Column 1" dataField="col1"/>
    <mx:DataGridColumn headerText="Column 2" dataField="col2"/>
    <mx:DataGridColumn headerText="Column 3" dataField="col3"/>
  </mx:columns>
</mx:DataGrid>

```

Clearly the **columns** property of the **DataGrid** can't be written as an attribute, because it expects a complex list of **DataGridColumn** objects, which in turn require their own property assignments.

S, FX, and MX: Namespaces Explained

Flex 4 provides three packages of components, and each is identified by a unique *namespace*. The three Flex 4 namespaces are Spark (**s:**), Halo (**mx:**), and the Language (**fx:**) namespace.

Each MXML component tag should contain a namespace designation, like the **Button** tag in Example 4-11, which declares the **Button** component from the Spark collection.

Example 4-11. An MXML Button in the Spark (s:) namespace

```
<s:Button/>
```

The Halo collection has its own **Button**, as in Example 4-12.

Example 4-12. An MXML Button in the Halo (mx:) namespace

```
<mx:Button/>
```

The same syntax is used to declare components from the Language (**fx:**) namespace, as shown in Example 4-13.

Example 4-13. An Array declaration in the Language (fx:) namespace

```

<fx:Declarations>
  <fx:Array id="statesArray">
    <fx:Object>Alabama, AL</fx:Object>
    <fx:Object>Alaska, AK</fx:Object>
    <fx:Object>Arizona, AZ</fx:Object>
    <fx:Object>Arkansas, AR</fx:Object>
  </fx:Array>
</fx:Declarations>

```

Namespace Basics

What's a namespace? Take a look at the word itself: *name + space*. Basically, a namespace is a package designation for a prebuilt set of components. You can illustrate this concept in Source mode by typing **<s:Bu**, and noticing that you have two **Button** options to choose between, as shown in Figure 4-4.

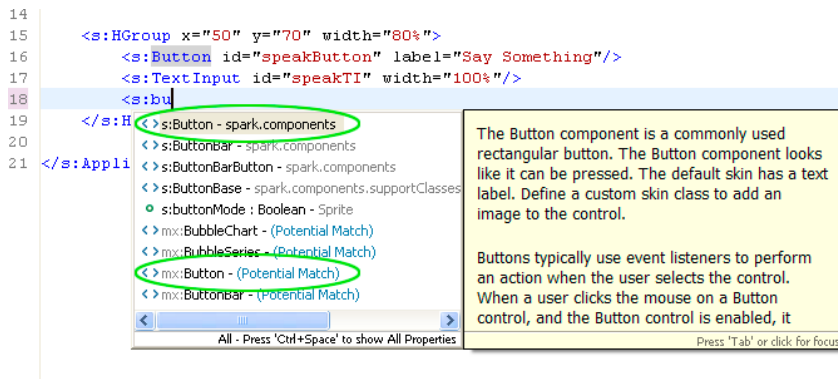


Figure 4-4. Button controls in the Spark (s:) and Halo (mx:) namespaces—the choice is yours

The **Button** class has been around for a while and continues to be important. Therefore, it exists among the new Spark components as well as the older Halo components. We recommend you use the updated Spark component, but because both packages are available within Flex 4, it's your choice. Similarly, you could create an extension of the Spark **Button** that includes a few specialized properties and save it in your own package/folder. To use the extended button in your project, you'd have to assign your custom package a namespace declaration. Custom namespaces are pretty common, even among small applications.

Most simply, namespaces point to component locations within a package structure. They may identify components in precompiled libraries (SWC files), or they may represent a classpath in your project's *src* folder.

If we revisit the concept of extending a **Button**, you could still name it "Button" because your unique namespace would distinguish it from its Spark/Halo counterparts, and the source code for your class would exist in a unique directory. You might then refer to your button in MXML as something like `<tweaks:Button/>` to differentiate it from `<s:Button/>` or `<mx:Button/>`.

Namespaces and Third-Party Libraries

If you work with a third-party library, code completion will add the library's namespace definition for you as you add components from the third-party library. For example, if you download the Google Maps API (it's a SWC) and add it to your *libs* folder, typing `<map` is enough for code hinting to lead you to `<maps:Map/>`, and in turn, Flash Builder will create the Google Maps namespace, as shown in Example 4-14.

Namespaces in Nested Properties

Notice that you must still use the namespace within a nested property of an MXML tag. For example, when assigning a property as a nested tag, you might be tempted to write the following incorrect code:

```

<s:Label>
  <text>Some Text</text>
</s:Label>

```

However, because the **text** property is a Flex component in its own right, it must include a namespace designation:

```

<s:Label>
  <s:text>Some Text</s:text>
</s:Label>

```

When using Flash Builder to write a Flex application, code completion will insert namespaces for you, so generally you won't have to think about this; however, third-party IDEs might not provide the same convenience.

NOTE

If you spend much time dabbling in third-party libraries, you'll frequently encounter the convention **com.domain.subpath.*** among package structures. The technique is called reverse domain package naming, and developers endorse it for two reasons. First, and most significantly, this strategy helps to keep component packages in unique classpaths, serving the purpose of avoiding class name conflicts. Second, well, there's a clear marketing advantage.

Example 4-14. A third-party library with a unique namespace identifier

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:maps="com.google.maps.*"
```

But what if you don't like that default namespace? Why not change **maps** to some other identifier? Well, you can.

Note the line **xmlns:maps="com.google.maps.*"**. This is easy to customize. For instance, instead of **maps**, you could use **gmap**, as demonstrated in Example 4-15.

Example 4-15. Changing a library's namespace

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:gmap="com.google.maps.*"
```

This simple change lets you use **<gmap:Map/>** instead of **<maps:Map/>** in your MXML component declarations.

Where Does This Namespace Point?

If you point your browser to **library://ns.adobe.com/flex/spark**, you won't get anything interesting. Although it looks like a path, it's really just an identifier. Using the web address paradigm helps ensure that you won't encounter package-naming conflicts. With this convention, you can create your own identifier using your web address, making it unlikely anyone else will use the same identifier.

Summary

In this chapter we explored the basics of Source mode. You learned how to clean up autogenerated code. You also learned how to add components and assign properties with the help of code completion. We discussed basic XML syntax rules, how to read XML structure, and how these relate to MXML. Finally, we discussed namespaces, giving you an improved perception of component package structure.

In the next chapter, we discuss the basics of scripting and object-oriented programming, and along the way, we'll finish that **HelloWorld** application.

ACTIONSCRIPT BASICS FOR FLEX APPLICATIONS

“A good chemist is twenty times as useful as any poet.”

—Ivan Turgenev

ActionScript is the glue that holds your application together. As you know, you'll use MXML to establish an application's layout and structure; in contrast, you'll use ActionScript to manage interaction, event handling, and processing logic.

Knowing where to place your script, how to create reusable code, and other scripting basics will help you build more powerful applications. Recognizing how ActionScript and MXML work together is key to understanding the Flex framework. In this chapter, you'll gain the knowledge you need to start making an impact with this powerful programming language.

Getting Ready

In this chapter, we'll finish the **HelloWorld** application by adding some interaction with ActionScript. So go ahead and open that project again; the code is shown in Example 5-1.

IN THIS CHAPTER

- Getting Ready
- Dot Notation
- Inline ActionScript
- Assignment and Concatenation
- Functions
- Variables
- Data Types
- Objects
- Classes
- ActionScript's Relationship with MXML
- Comments?
- Summary

Example 5-1. The *HelloWorld* application, going into the final round

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  width="100%" height="100%">

  <s:Label id="titleLabel" x="50" y="50"
    fontFamily="Arial" fontWeight="bold"
    text="My First Flex 4 Application"/>

  <s:HGroup x="50" y="70" width="80%">
    <s:Button id="speakButton" label="Say Something"/>
    <s:TextInput id="speakTI" width="100%" />
    <s:Button id="clearButton" label="Clear"/>
  </s:HGroup>

</s:Application>
```

Naming Conventions

Adobe recommends getting in the habit of using fully-qualified component identifiers and variable names. In **HelloWorld**, both **speakButton** and **clearButton** are examples of this approach.

This naming convention pairs a succinct one- or two-word description with a class name using a convention known as **CamelCase**. There are two varieties of CamelCase: lowerCamelCase and UpperCamelCase. The preferred habit is to use lowerCamelCase for variable instances (we'll discuss variables later in the section titled "Variables") and UpperCamelCase for class names.

This approach to naming helps define both the purpose and the data type of a component or a variable. Some developers might criticize this approach (and their good reasons continue to remain elusive), but it produces code that is much more intuitive, particularly for anyone tasked with maintaining it months later.

Another approach we endorse uses abbreviations for long class types. For instance, **ArrayCollection** could be abbreviated **AC**. From the **HelloWorld** example, **speakTI** demonstrates this approach, with a short, purposeful name followed by the abbreviated class type, in this case **TI** for **TextInput**.

HelloWorld has two buttons: **speakButton**, which we created in Design mode, and **clearButton**, which we created in Source mode. Often, buttons are the main point of contact for making something happen, so that's where we'll start.

We'll use the **click** event on **speakButton** to populate the **TextInput** with a greeting. Fortunately, we already handled some of the groundwork in Chapter 3 when we assigned each component an **id** property.

In order to refer to containers and components with code, each component being referenced needs to have a unique identifier (**id**) property. It's a good habit to give all your significant components a unique **id** as you create them; after all, component identifiers help keep your code readable.

The **HelloWorld** application currently includes four unique components: **titleLabel**, **speakButton**, **speakTI**, and **clearButton**. We'll be working with the latter three.

Dot Notation

Like many programming languages, **ActionScript** provides a convenient syntax for accessing classes and their properties, called *dot notation*. Dot notation works like this:

```
componentId.componentProperty = someValue;
```

Or closer to home:

```
speakTI.text = "Hi I say!";
```

In this example, the **TextInput** component's **text** property is assigned the value of "Hi I say!". Alternatively, we can clear the **text** property by setting it equal to an empty string:

```
speakTI.text = "";
```


This chapter demonstrates how to handle such tasks with ActionScript, and we discuss two varieties of scripting: inline ActionScript and the Script/CDATA block. First, let's consider inline ActionScript.

Inline ActionScript

Script placed inside an MXML tag attribute is known as *inline ActionScript*. Flex UI components can listen for and respond to certain events, such as clicking, dragging the mouse, or typing with the keyboard. A typical scenario involves a user clicking a **Button**, with the **click** event triggering a scripted reaction.

Let's demonstrate inline ActionScript by programming an inline **click** event for **speakButton** to populate our **TextInput** with a greeting, shown in Example 5-2.

Example 5-2. Adding inline script for a Button click event

```
<s:Button id="speakButton" label="Say Something"
  click="speakTI.text = 'Hi I say!'" />
```

All you had to do was add a **click** attribute to the button, and within that attribute, include some simple ActionScript. That's scripting at its easiest, placed right in an MXML tag.

But what if you wanted to handle two or more actions when someone clicks a button? Conveniently, you can establish multiple statements within the **click** attribute.

This time we'll use **speakButton** to post our greeting as well as change the title of our application (**titleLabel.text**) to "My Flex Application, Hard at Work". So modify your code to include a little more in the button's **click** attribute, as shown in Example 5-3.

Example 5-3. Two lines of inline script, separated by a semicolon (;)

```
<s:Button id="speakButton" label="Say Something"
  click="speakTI.text = 'Hi I say!';
  titleLabel.text = 'My Flex Application, Hard at Work'" />
```

Notice a semicolon (;) separates the two script assignments. The semicolon tells the compiler you have multiple statements occurring on the same line.

A downside of this technique is that it can lead to messy code; if you had three or four script actions, the **click** attribute would become a monstrosity. The truth is, we don't find many examples like this out in the wild. Fortunately there's another way to apply ActionScript, and that's from within a Script/CDATA block, which we cover later in the section titled "Functions" on page 66.

NOTE

This example uses single quotes ('), and not double quotes (") to enclose the greeting message because two pairs of double quotes within the attribute definition would confuse the compiler. For more information, see the sidebar titled "Double Take" on page 66.

Assignment and Concatenation

Setting the value of a property involves *assignment*, which is accomplished using the equals sign (=) followed by the desired value. We just demonstrated assignment in the previous example:

```
speakTI.text = "Hi I say!";
```

NOTE

There are a tremendous number of operators in ActionScript, of which most books cover only a select few. If you want to see the full list for yourself, follow this link to Adobe's ActionScript 3.0 Language Reference; it's the kind of link you'll want to bookmark: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/operators.html.

When used in a string/text context, like we show here, the equals sign is called the *concatenation operator*, but if it's being used in a mathematical context, the equals sign is called the *addition operator*. While surely you're familiar with addition, *concatenation* is the programming term for "adding text together." This would be an example of concatenation:

```
speakTI.text = "Hi" + " " + "Bill" + "!";
```

Notice that every nuance of the phrase, including the space and the punctuation, is accounted for in the expression. We wrote it that way to exaggerate the idea, but the following would be more typical in practice, only with a variable replacing the hardcoded name value:

```
speakTI.text = "Hi " + "Bill" + "!";
```

When property values are assigned within an MXML attribute, they're set as soon as the component is created. With ActionScript, though, you have more control over timing.

Double Take

For the inline ActionScript example, you probably noticed the use of single quotes (') instead of double quotes (") within the **click** attribute; however, under the Assignment heading, we returned to double quotes.

In the first example, single quotes were necessary because the Flex compiler expects double quotes to open and close the **click** event's definition. If you prefer, you can switch between single and double quotes, so long as the quote opening the assignment also closes it, for example:

```
<s:Button click='speakTI.text = "Hi I say!"/>
```

Functions

A *function* is code you write to handle various processing tasks. You place some ActionScript code within a function, give the function a name, and when you want to run that code, you reference that function.

Sometimes you'll call functions from attribute *events* using inline ActionScript. But you can also call functions from within other functions. Simple functions, such as the sort we're about to see, require no dynamic input. On the other hand, complex functions may accept passed-in values, manipulate those values, and return them elsewhere.

In the following section, we demonstrate the second type of scripting by creating a special function inside a Script/CDATA block to clear our **TextInput** control.

Where to Place a Function

ActionScript functions should be placed inside a Script/CDATA block, which can be created only in Source mode.

To create a **Script** block, place your cursor somewhere between the opening Application tag and your UI components and type **<fx:script**. That should be enough for code completion, so hit Tab and then Enter/Return to commit the code. Next, close the opening script tag by adding a right angle bracket (**>**).

It's good practice to keep your script near the top of your application code. Note the placement of the Script/CDATA block in the context of Example 5-4.

NOTE

ActionScript functions go inside a Script/CDATA block.

Example 5-4. Adding a Script/CDATA block for ActionScript functions

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  width="100%" height="100%">

  <fx:Script>
    <![CDATA[

    ]]>
  </fx:Script>

  <s:Label id="titleLabel" x="50" y="50"
    fontFamily="Arial" fontWeight="bold"
    text="My First Flex 4 Application"/>

  <s:HGroup x="50" y="70" width="80%">

    <s:Button id="speakButton" label="Say Something"
      click="speakTI.text = 'Hi I say!';
      titleLabel.text = 'My Flex Application, Hard at Work'"/>

    <s:TextInput id="speakTI" width="100%" />

    <s:Button id="clearButton" label="Clear" />

  </s:HGroup>

</s:Application>
```

You might be curious about that **CDATA** tag. **CDATA** is a special XML entity that tells the XML compiler to ignore the contents of the tag. The ActionScript you'll write here might have characters such as quotes and angle brackets (**<** and **>**) that could confuse the XML parser, so you wrap it in a **CDATA** block.

Don't worry about remembering how to script a **CDATA** tag, because Flash Builder's code completion will finish the block automatically.

It's within the Script/CDATA block that we'll add named ActionScript functions, which we learn how to do next.

How to Create a Function

To create a function within a Script/CDATA block, use the keyword **function** followed by a descriptive name, followed by a couple of parentheses, followed by a left brace (**{**), which opens the function block. When you hit Enter following the left brace, Flash Builder will automatically close the function block by adding a carriage return and the right brace (**}**). For example:

```
<fx:Script>
  <![CDATA[

      function clearSpeakTI(){

      }

  ]]>
</fx:Script>
```

What's the meaning of those empty parentheses? If we needed to pass some variables into this function, we would reference those variables within the parentheses. In this case, we're just clearing a **TextInput** control—we're not processing any data—so there's no need to pass in any variables. We will, however, see some examples of this later in the section titled "Function Parameters" on page 72.

Looking back at the construction of the function, the left and right braces (**{}**) frame the function block. Since we have a framed function ready and willing, let's give it some work. In one line, set the **text** property of **speakTI** equal to an empty string (**""**). Then, in a second line, reset the label text back to "My First Flex 4 Application", like so:

```
<fx:Script>
  <![CDATA[

      function clearSpeakTI(){
        speakTI.text = "";
        titleLabel.text = "My First Flex 4 Application";
      }

  ]]>
</fx:Script>
```

Now all you have to do is call the function from somewhere in order to clear the **TextInput**. So how do you call a function?

In this case, we'll call our new function using inline ActionScript from the **clearButton** component's **click** event. It's easy; just modify **clearButton** to include the following addition:

```
<s:Button id="clearButton" label="Clear" click="clearSpeakTI()"/>
```

Altogether, this should give you the application code shown in Example 5-5.

Example 5-5. *HelloWorld, using both inline and block script*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%">

    <fx:Script>
        <![CDATA[

            function clearSpeakTI(){
                speakTI.text = "";
                titleLabel.text = "My First Flex 4 Application";
            }

        ]]>
    </fx:Script>

    <s:Label id="titleLabel" x="50" y="50"
        fontFamily="Arial" fontWeight="bold"
        text="My First Flex 4 Application"/>

    <s:HGroup x="50" y="70" width="80%">
        <s:Button id="speakButton" label="Say Something"
            click="speakTI.text = 'Hi I say!';
                titleLabel.text = 'My Flex Application, Hard at Work'"/>
        <s:TextInput id="speakTI" width="100%"/>
        <s:Button id="clearButton" label="Clear"
            click="clearSpeakTI()"/>
    </s:HGroup>

</s:Application>
```

When the **clearButton** is clicked, our function, **clearSpeakTI()**, will be called and **speakTI** will be cleared, effectively resetting the control. Now run the program to see it in action (Figures 5-1 and 5-2).

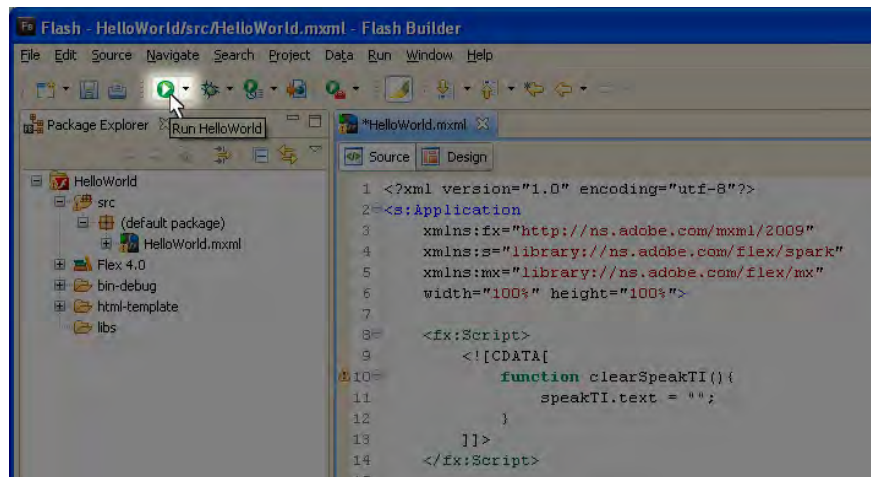


Figure 5-1. Running the HelloWorld application

My Flex Application, Hard at Work



Figure 5-2. Interacting with the application

Function Access

If you programmed **HelloWorld** exactly as we presented it, have you noticed Flash Builder trying to alert you to some shortcomings in the code (Figure 5-3)?



Figure 5-3. Inspecting the IDE warnings

Flash Builder is taking exception to two issues we can call errors of omission. In other words, the compiler expected us to include some more code, and it's worried.

The first warning states the return value for our function has no type declaration:

1008: return value for function 'clearSpeakTI' has no type declaration.

We'll come back to that one a little later, so just ignore it for now. The second warning informs us that our function is not formally "scoped":

1084: function 'clearSpeakTI' will be scoped to the default namespace: HelloWorld: internal. It will not be visible outside of this package.

We'll fix the scoping issue first.

A function's *scope* refers to its level of availability to other components in an application. Scope is established using keywords called *access modifiers*, which include **public**, **private**, **protected**, and **internal**. Access modifiers are placed ahead of the **function** keyword to define the function's scope. Let's consider the four types of scope:

NOTE

Scope and access level also apply to variable declaration, which we discuss a little later in this chapter.

public

Functions declared in the **public** scope can be accessed by any class in the project.

private

The **private** scope identifies functions that are invisible beyond their defining component; private functions are useful only to the class where they are defined.

internal

The **internal** scope identifies functions that are accessible to any class in the same package, or folder, as the class defining the function. If a function is undefined, Flash Builder compiles it with **internal** scope; this is the default.

protected

Functions having **protected** scope are available only to the defining class and any subclasses of the defining class.

Assuming you still have **HelloWorld** open, return to the function **clearSpeakTI()** and set it to **private** scope. Since we know **clearSpeakTI()** will not be called by any other components in our application, limiting its scope is better practice. Example 5-6 shows what you're going for.

Example 5-6. Restricting a function to the private scope

```
private function clearSpeakTI(){
    speakTI.text = "";
    titleLabel.text = "My First Flex 4 Application";
}
```

Proper scoping allows you to strategically expose or conceal functions of one class to other classes in an application. Scoping matters most when you're developing a large project, extending components, or creating custom packages. Nevertheless, it's still wise to observe correct scoping techniques. As a

rule of thumb, if you expect to call a function from other components, then either the **public** or **internal** scope will be necessary; otherwise, the **private** scope is more appropriate.

If you're getting a lot of errors, or if you're just not sure which scope to use, try setting the scope of your functions to **public** for the meantime. The public scope makes a function available anywhere in your application. You can always restrict the scope later. This does require noting, however, that creating every function in the **public** scope is bad practice, and we're offering this thought as a contingency more than a strategy.

Returning to our example, setting the scope of `clearSpeakTI()` to **private** should remedy the first compiler warning, and if you save, the warning should disappear.

Next we discuss passing parameters to and from functions. We'll return to our other compiler warning later in the section titled "Data Types" on page 75.

Function Parameters

Remember those parentheses? We previously mentioned they're used to pass variable information, called *parameters*, into a function. Many functions won't require any data inputs, and those parentheses will remain empty; however, some functions specify several required inputs, and when you call those functions, you'll need to pass in some parameters.

Say you wanted to tell a function what text to assign to a **TextInput** control. You could do that by passing a parameter directly into a function. To demonstrate, add the function shown in Example 5-7 inside your Script/CDATA block.

Example 5-7. Building a function to accept an input parameter

```
private function postGreeting(name:String){
    speakTI.text = "Hi " + name + "!";
}
```

NOTE

On the heels of our scoping discussion, local scope refers to variables that are instantiated, consumed, and ultimately destroyed by their defining function.

The code in Example 5-7 creates a **String** variable called **name** in a function's local scope. When you call this function, it expects you to pass it a **String** value, within parentheses, that it will assign to its **name** variable. To call this function, modify the inline **click** event for **speakButton** as shown in Example 5-8.

Example 5-8. Calling a function and passing it a value using inline script

```
<s:Button id="speakButton" label="Say Something"
    click="postGreeting('Bobby');
    titleLabel.text = 'My Flex Application, Hard at Work'"/>
```

You can also create functions that accept multiple parameters, such as the one in Example 5-9.

Example 5-9. Building a function to accept two input parameters

```
private function postGreeting(firstName:String, lastName:String){
    speakTI.text = "Hi " + firstName + " " + lastName + "!";
}
```

And you would call that function like Example 5-10.

Example 5-10. Passing two parameters into a function

```
<s:Button id="speakButton" label="Say Something"
    click="postGreeting('Bobby','Bodico');
    titleLabel.text = 'My Flex Application, Hard at Work'"/>
```

Sometimes it's useful to assign default values to the parameters you're defining for a function. Planning ahead for default values gives you the option of calling the function without passing in parameters—if you don't pass in a value, the parameter will use its default value. You set defaults by establishing a parameter value in the function declaration, as in Example 5-11.

Example 5-11. Assigning default values to function parameters

```
private function postGreeting(firstName:String = "Guest",
    lastName:String = ""){
    speakTI.text = "Hi " + firstName + " " + lastName + "!";
}
```

Since you've assigned defaults, you can call the function with zero, one, or two parameters. Example 5-12 shows a function call with none.

Example 5-12. Calling a function to use default parameters

```
<s:Button id="speakButton" label="Say Something"
    click="postGreeting()"/>
```

Example 5-13 has one, overriding the default **firstName** value of "Guest".

Example 5-13. Calling a function to override one of two default parameters

```
<s:Button id="speakButton" label="Say Something"
    click="postGreeting('Bobby')"/>
```

And Example 5-14 has two, overriding default values for **firstName** and **lastName**, respectively.

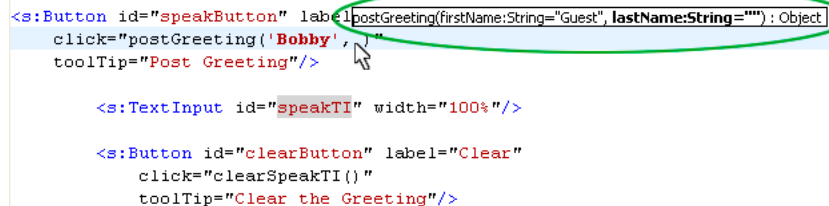
Example 5-14. Calling a function to override two of two default parameters

```
<s:Button id="speakButton" label="Say Something"
    click="postGreeting('Bobby', 'Bodico')"/>
```

After you've programmed a function to handle parameters, any time you start writing code to call that function, code completion will sense the required parameters and provide you with a nice pop up indicating the order of required parameters, their data types, and any default values they may assume. Figure 5-4 illustrates this feature in action. Note that the first parameter has already been entered, and the second parameter—which remains to be addressed—is highlighted in bold text.

NOTE

These examples are using concatenation to build a text string in piecemeal fashion. We discussed concatenation previously in the section titled "Assignment and Concatenation." Notice that we accounted for a space (" ") between the **firstName** and **lastName** values in this welcome message.



```
<s:Button id="speakButton" label="Post Greeting" click="postGreeting(firstName:String='Guest', lastName:String='') : Object" tooltip="Post Greeting"/>

<s:TextInput id="speakTI" width="100%" />

<s:Button id="clearButton" label="Clear" click="clearSpeakTI()" tooltip="Clear the Greeting"/>
```

Figure 5-4. Passing parameters into the `postGreeting()` function

Variables

Variables are just in-memory “containers” that represent data values, like numbers, text, or instances of more complex objects such as arrays, XML, UI components, and so on. If you wanted to store someone’s name and recall it elsewhere in an application, you would create a **String** variable and assign the individual’s name as the value of that variable. Let’s see an illustration of this in Example 5-15.

NOTE

The phrase “data type assignment” refers to code declaring the class or fundamental data type a variable will instance. For example, a variable can be “typed” as something simple, such as a **String** or a **Number**; alternatively, a variable can be typed as a complex object, such as an **Array**, **XML**, or even a **Flex UI component**.

Example 5-15. Declaring a *String* variable and assigning it a value

```
var username:String;
username = "Elijah";
```

In this example, the **var** keyword indicates that we’re creating a variable, and it’s followed by the descriptive variable name, **username**. Immediately after the variable name is a colon (:), also called the *type operator*, which tells the compiler that a data type assignment is coming. The colon is followed by the desired data type—**String** in this case. Finally, a semicolon (;) terminates the line of code.

The second line of code assigns a text value, **Elijah**, to the new **String** variable.

You also have the option of creating a variable and setting its default value in the same line of code, as shown in Example 5-16.

Example 5-16. Declaring a variable and assigning its value in one line

```
var username:String = "Elijah";
```

Also, just like with functions, you can set the scope of your variables, as in Example 5-17.

Example 5-17. Declaring a variable in the public scope

```
public var username:String = "Elijah";
```

NOTE

Variable scope is established only for application- and component-level variables declared near the top of a `Script/CDATA` block. Variables declared within the body of a function, however, should not receive scope assignments. In fact, if you try to set the scope of a function variable, Flash Builder will complain.

Data Types

In programming, *data typing* means explicitly stating which class or data type a variable will represent. Proper typing improves application performance, and stringent typing habits force developers to ponder what variety of information will be needed in a particular situation.

Fundamental Data Types

You'll use some fundamental data types regularly, and these are described in Table 5-1. At first, refer back to this table as necessary, and sooner rather than later, you'll become a pro at data typing.

Table 5-1. *Fundamental data types*

Name	Description	Example	Default value
<code>String</code>	Text, plain and simple. Can be one character or many. String is short for “string of characters.”	<code>var hi:String = "Hello!";</code>	<code>null</code>
<code>Number</code>	A numeric value that can be a fraction (decimal value).	<code>var pi:Number = 3.14;</code>	<code>NaN</code> ("Not a Number")
<code>uint</code>	An “unsigned” integer—a whole number that can't be negative. Can be in the range from 0 to 4,294,967,295 .	<code>var lightSpeedMPS:uint = 299792458;</code>	<code>0</code>
<code>int</code>	Any integer (a whole number, no fractions or decimals). Can be in the range from -2,147,483,648 to 2,147,483,647 .	<code>var neg:int -12</code>	<code>0</code>
<code>Boolean</code>	A true/false value, like a switch. Valid values are true and false .	<code>var isHappy:Boolean = true</code>	<code>false</code>
<code>void</code>	A special value for functions, meaning the function returns nothing. The only value it can have is undefined .	<code>function doNothing(): void { } }</code>	<code>undefined</code>

Say your application has a form that asks for user information, such as name and age. The name, which is composed of text characters, is a **String**. The age would need to be a numeric value—but which numeric value? Well, age will always need to be a positive whole number, so looking at Table 5-1, you can see that the data type `uint` would be a good choice, but both `int` and `Number` would also work.

Coercion, Casting, and Conversion

Sometimes you'll need to convert from one data type to another.

A common example involves attempting to assign a numeric value to the **text** property of a **TextInput**. Implying in code that a variable of one type should be handled as another type is called *coercion*, or *implicit coercion*, and it generally produces errors.

More often than not, you'll need to perform an explicit conversion, called *casting*, to handle the variable as intended. Casting involves telling the compiler to convert between two data types, such as changing a numeric value to a **String** value.

One approach to casting uses the constructor of the desired type to convert the value. As we'll see in the later section discussing classes, the *constructor* is the special method used to create an instance of a class, such as **String()**. Creating a constructor and passing it a value will attempt to convert between the value types:

```
textInput.text = String(numericStepper.value);
```

Another approach involves using the **toString()** method, which most classes inherit. The **toString()** method returns a **String** representation of a value:

```
textInput.text = numericStepper.value.toString();
```

Typing Variables

The previous section demonstrated how to build a variable declaration. So, to type a variable called **age** as a **uint**, you would use the code in Example 5-18.

Example 5-18. Declaring a uint variable

```
var age:uint = 30;
```

All you did here was insert a colon followed by the data type **uint**. This syntax applies to every manner of data typing. The code in Example 5-19 creates a variable called **pi** and types it as a **Number**.

Example 5-19. Declaring a Number variable

```
var pi:Number = 3.14;
```

Typing Functions

For functions, typing is similar. We haven't yet discussed this aspect of functions, but functions have the ability to *return* values, and any value they return should be typed. To type the return value of a function, follow the parentheses with a colon and the proper data type, just as you would for a variable; then, inside the function block, dispatch the processed value using the **return** keyword. The function in Example 5-20 returns the sum of 2 and 2.

Example 5-20. *Typing a function to return a Number value*

```
public function doSomeMath():Number{
    return 2 + 2;
}
```

To use the product of such a function in code, you simply call the function where you want to apply its value. Example 5-21 would assign the product of the function to the **Number** variable **myMath**.

Example 5-21. *Assigning the product of a function to a variable*

```
var myMath:Number = doSomeMath();
//the value of myMath would be 4
```

Remember earlier when we noticed two problems with our **clearSpeakTI()** function? We can finally address the first problem, which stated:

1008: return value for function 'clearSpeakTI' has no type declaration.

It's easy to fix. Instead of leaving that function untyped, just append the function declaration to return the **void** data type, as in Example 5-22.

Example 5-22. *A function that won't return a value should be typed as void*

```
private function clearSpeakTI():void{
    speakTI.text = "";
    titleLabel.text = "My First Flex 4 Application";
}
```

The function doesn't return anything, so the **void** data type is added as a courtesy to the compiler saying, "This function won't return anything, so don't expect it to."

Back in **HelloWorld**, you'll want both **clearSpeakTI()** and **postGreeting()** to be typed as **void**, and with that accomplished, our **HelloWorld** program is finally done, dunzo, finished even! Example 5-23 shows the final application code for **HelloWorld**, and Figure 5-5 shows the application in action.

Example 5-23. *The completed HelloWorld application*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    width="100%" height="100%">

    <fx:Script>
        <![CDATA[

            private function clearSpeakTI():void{
                speakTI.text = "";
                titleLabel.text = "My First Flex 4 Application";
            }

        ]]>
    </fx:Script>
</s:Application>
```

NOTE

To assign a function's return value to a variable, both the function and the variable must be of the same data type.

```

        private function postGreeting(firstName:String,
        lastName: String):void{

            speakTI.text = "Hi " + firstName + " " + lastName + "!";

        }

    ]]>
</fx:Script>

<s:Label id="titleLabel" x="50" y="50"
fontFamily="Arial" fontWeight="bold"
text="My First Flex 4 Application"/>

<s:HGroup x="50" y="70" width="80%">
    <s:Button id="speakButton" label="Say Something"
        click="postGreeting('Bobby', 'Bodico');
            titleLabel.text = 'My Flex Application, Hard at Work'"/>
    <s:TextInput id="speakTI" width="100%"/>
    <s:Button id="clearButton" label="Clear"
        click="clearSpeakTI()"/>
</s:HGroup>

</s:Application>

```

My Flex Application, Hard at Work

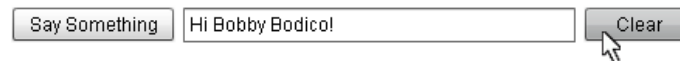


Figure 5-5. The completed HelloWorld application, in its farewell salute

We're closing the book on our first program, but we're not yet done with the chapter. We'll spend the rest of the time discussing object-oriented programming and the relationship between ActionScript and MXML.

Objects

Everything is an *object*. That's right, in an object-oriented programming (OOP) language such as ActionScript, everything you encounter is an object of some kind; that's why you'll notice the word "object" mentioned throughout this text. "But that still doesn't tell me what an object is!" you say? OK, here goes.

An object is a generic container for anything else. It can hold a piece of text or some numbers, it can have logic that manipulates pieces of data, and it can even hold other objects. It can be thought of as something that has both state and behavior, meaning it can have variables (state) and methods that work with those variables (behavior). It's the basic building block of any Flex application because, again, everything is an object. Your application is an object. The buttons inside your application are objects, and any variables you create are objects.

We'll show you how you can create your own object. Say you want to create a new object, **car**, and give it some properties that you expect a car to have, such as **type** and **color**. You can do that in ActionScript with the following code, first creating an instance of an **Object** and then giving it properties:

```
var car:Object = new Object();
car.type = "sports car";
car.color = "yellow";
car.topSpeed = 170;
car.isInsured = false;
car.driver = undefined;
```

We can also create a **driver** for this car:

```
var person:Object = new Object();
person.name = "Grant";
person.age = 25;
```

Now you can modify the **driver** property of the **car** object to include the **person**:

```
car.driver = person;
```

Voilà! Now you have a fast sports car with no insurance driven by 25-year-old Grant. You're living dangerously! But, what if you expect this program to have several cars and drivers? For this scenario, custom classes would make a better solution.

Classes

If everything is an object, then what is a class? A *class* is like a blueprint for an object. Classes define the properties and methods an object will have. Accordingly, a *class variable* is just an in-memory container representing an instance of a class.

If you want to follow along with the examples in this section, you may want to create a new project in your workspace. We called this project **ClassTesting**.

Packages

ActionScript classes expect to belong to a particular package. A *package* is simply a unique folder structure that stores source code files for various classes. Flash Builder will allow you to maintain a package anywhere in your filesystem, so we'll take advantage of that.

Using whatever method you prefer, create the following folder structure in your workspace; again, we're showing `C:\Flex4Code\` as the workspace:

```
C:\Flex4Code\com\learningflex4\classes
```

This means your workspace should include a folder named *com*, which includes the folder *learningflex4*, which includes the folder *classes*.

NOTE

*Flash Builder won't provide code completion for **Object** classes, because there is no underlying definition of the properties or methods available to an **Object**. To get code completion, you'll need to create a class, which we discuss in the next section.*

NOTE

By convention, package names should be all lowercase.

With a folder structure set up for our custom package and a new project ready to go, we're ready to link the package into our project. Make sure the project is selected in the Package Explorer, and choose Project→Properties→Flex Build Path. Then, click the “Source path” tab, followed by the Add Folder button.

In the Add Folder dialog, browse to the *com* folder in your workspace and add it (Figure 5-6).

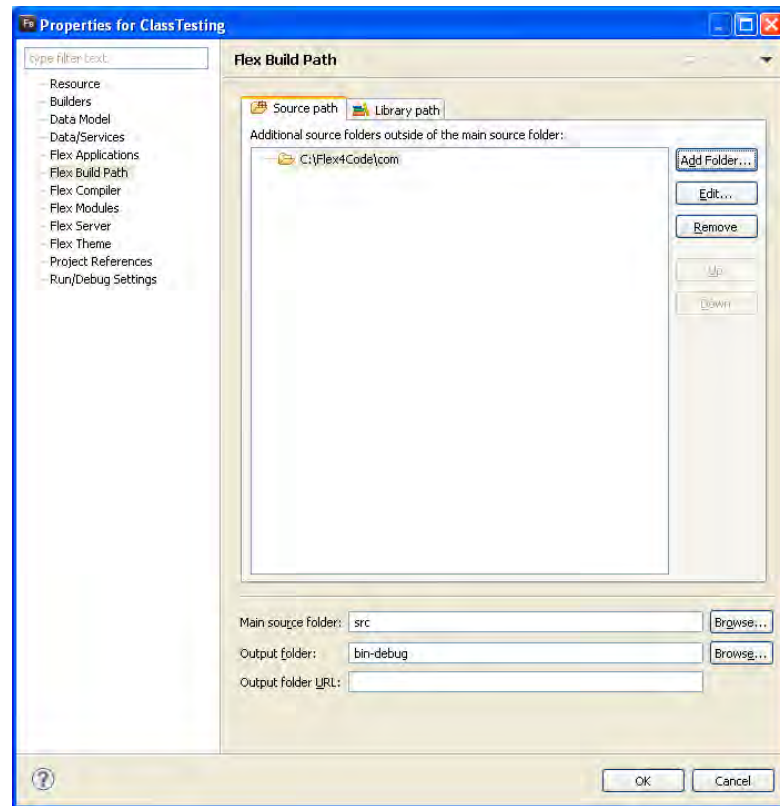


Figure 5-6. Adding a package as an additional source path

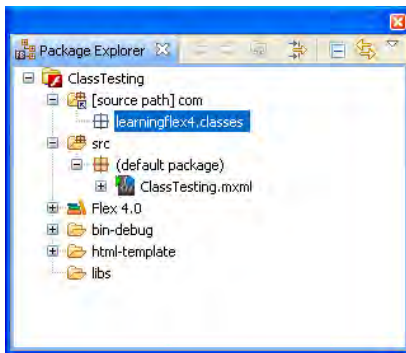


Figure 5-7. An externally maintained package added to the project

NOTE

Flash Builder will allow you to start a new class name with a lowercase letter, but convention dictates that class names should start with an uppercase letter and use UpperCamelCase.

That should link the package to your project and simultaneously create a new source path in the Package Explorer (Figure 5-7).

Now we can start adding custom classes to the package.

Making a New Class

To create a new class, first select the external package in the Package Explorer, and then choose File→New→ActionScript Class.

When the dialog appears, it should already have the correct package name, *learningflex4.classes*. Now name the class **Car** and click Finish (Figure 5-8).

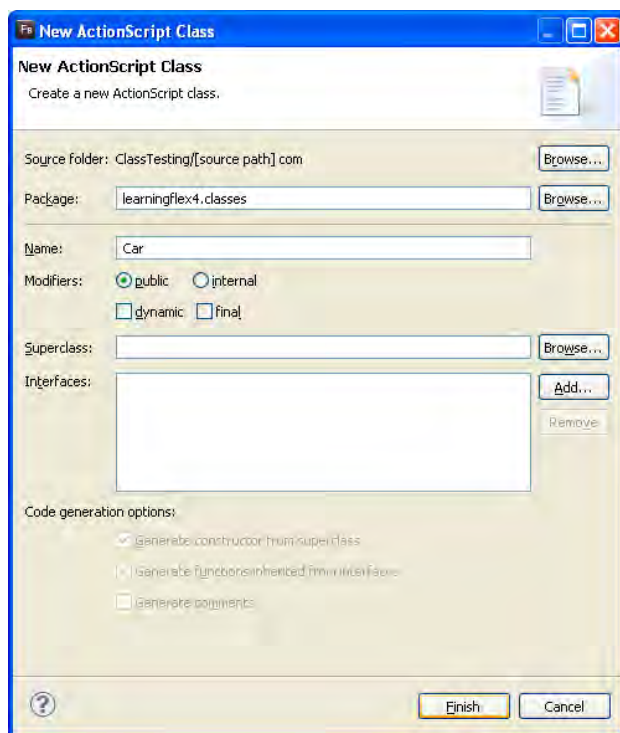


Figure 5-8. Adding a class with the new class dialog

When you emerge into the editor, you'll have the empty class code shown in Example 5-24.

Example 5-24. *C:\Flex4Code\com\learningflex4\classes\Car.as*

```
package learningflex4.classes
{
    public class Car
    {
        public function Car()
        {
        }
    }
}
```

The portion in the middle—**public function Car()**—is the class's *constructor* method. A class's constructor handles anything that needs to happen when the class is initialized for use in an application. Examples could include assigning default property values or even initializing other classes. Since we don't have a complicated class, we don't need to handle anything within the constructor.

Class Properties

Class properties are variable values defined inside a class definition. As with our previous discussion of variables, class variables should have scope, which affects their visibility within and outside of the class. The easiest way to give a class externally accessible properties is to declare its variables in the **public** scope.

Example 5-25 illustrates how we would give our **Car** class the same properties as the **car** object.

NOTE

The properties in this class are strongly typed, meaning you can expect the **type** to be a **String**, **color** to be a **uint**, and **isInsured** to be a **Boolean**. You even know that the **driver** property is a **Person**. Strongly typed values are helpful for both you and the Flex compiler.

Example 5-25. Defining property variables for the *Car* class

```
package learningflex4.classes
{
    public class Car
    {
        public var type:String;
        public var color:uint;
        public var topSpeed:int;
        public var isInsured:Boolean;
        public var driver:Person;

        public function Car()
        {

        }

    }
}
```

Now that you know how to make a class and give it some properties, review those steps by making the **Person** class, which goes in the same package as **Car**, illustrated in Example 5-26.

Example 5-26. C:\Flex4Code\com\learningflex4\classes\Person.as

```
package learningflex4.classes
{
    public class Person
    {
        public var name:String;
        public var age:uint;

        public function Person()
        {

        }

    }
}
```

NOTE

A more formal manner of handling class properties is through the use of so-called “Getter” and “Setter” functions. If you'd like to read ahead and examine this approach, check out the box titled “Getters and Setters” on page 357 in Chapter 16.

Class Methods

It should be easy to understand methods, because methods are functions. As with class variables, class functions have scope; some functions may be accessible outside of the class, and other functions will be accessible only within the class.

Example 5-27 gives the **Car** class a public method (**drive**) that accepts a starting place (**placeA**) and a destination (**placeB**), and returns a formatted string of text.

Example 5-27. Defining a method for the *Car* class

```
package learningflex4.classes
{
    public class Car
    {
        public var type:String;
        public var color:uint;
        public var topSpeed:int;
        public var isInsured:Boolean;
        public var driver:Person;

        public function Car()
        {

        }

        public function drive(placeA:String, placeB:String):String
        {
            return ("starting at: " + placeA + "\n" +
                "going to: " + placeB);
        }
    }
}
```

NOTE

In this snippet, `\n` stands for “new-line,” and it will be interpreted as a line break by many controls, including the **TextArea**, which we’re about to see. If you prefer, you could replace `\n` with `\r`, which stands for “carriage return,” as both entities are treated identically.

Working with Custom Classes

Now that we’ve created both the **Car** and **Person** classes, let’s use them in an application. The code in Example 5-28 for **ClassTesting** uses two events to make things happen.

First, the **Application** container’s **creationComplete** event calls the function **makeCarAndDriver()**, which assigns values to the **car** and **person** variables that are declared in the application’s private scope.

Second, the **Button** control’s **click** event calls the function **hitTheRoad()**, which outputs this information into a **TextArea** control.

This example uses a **TextArea** control to render our output, and we’re introducing you to a couple of tricks here. First, we’re using the *newline entity* (`\n`) to represent line breaks in the output, and this entity needs to be wrapped in quotes and concatenated into the text string. Also, we’re using the *concatenation assignment operator* (`+=`) to append text to the **TextArea**. If we used the regular *concatenation operator* (`=`), each new text string would fully replace the one that preceded it.

Tit for Tat

The import statements in Example 5-28 will be added automatically by Flash Builder as you create their variable instances. That means, typing `private var car:Car` will be enough for Flash Builder to add `import learningflex4.classes.Car;` near the top of the Script block. You can manually enter the import statements if you prefer, but you shouldn't have to.

On the contrary, if you copy and paste code into Flash Builder, it will not add required import statements; in that case you'll need to manually add import statements. This is often encountered when copy/pasting code from blog tutorials, etc., into an application, after which, Flash Builder will throw an error. So if you've copied and pasted code and you're getting an error, make sure you have all the necessary import statements.

Go Somewhere!

```
car: sports car
driver: Grant
travel speed: 75
starting at: Elijah's place
going to: VillaGIS
```

Figure 5-9. Working with custom classes

Example 5-28. The application code for ClassTesting

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  creationComplete="makeCarAndDriver()">

  <fx:Script>
    <![CDATA[
      import learningflex4.classes.Car;
      import learningflex4.classes.Person;

      private var car:Car = new Car();
      private var person:Person = new Person();

      private function makeCarAndDriver():void{
        car.type = "sports car";
        car.color = 0x000000;
        car.topSpeed = 75;
        car.isInsured = false;

        person.name = "Grant";
        person.age = 25;

        car.driver = person;
      }

      private function hitTheRoad():void{
        textArea.text = "car: " + car.type + "\n";
        textArea.text += "driver: " + car.driver.name + "\n";
        textArea.text += "travel speed: " + car.topSpeed + "\n";
        textArea.text += car.drive("Elijah's place", "VillaGIS");
      }
    ]]>
  </fx:Script>

  <s:VGroup horizontalCenter="0" verticalCenter="0">
    <s:Button label="Go Somewhere!" click="hitTheRoad()"/>
    <s:TextArea id="textArea"/>
  </s:VGroup>

</s:Application>
```

To see how it all works, go ahead and run it. You should end up with something like Figure 5-9.

Knowing how to create custom classes will be helpful as you gain more knowledge of Flex. It will not only help you understand the foundations of the framework, but when you start extending the base components, you'll have a better impression of the bigger picture, which is the object-oriented programming behind the scenes.

ActionScript's Relationship with MXML

As we've seen, MXML is great for laying out structure, and ActionScript is suited for defining method logic and interaction. So what is the difference between the two? In many ways they're the same thing.

MXML and ActionScript Work Together

MXML and ActionScript complement each other. You'll find it's easier to build an application's structure using MXML than using just ActionScript. MXML unfolds faster, it's easier to read and write, and you get results from fewer keystrokes. But MXML doesn't replace ActionScript. Scripting is quite necessary for serious Flex applications. In fact, if anything it's MXML that's unnecessary, because if you wanted, you could build a full application using just ActionScript.

MXML Becomes ActionScript

When you compile an application, all the MXML you created is initially interpreted into ActionScript, and then it's processed into a binary SWF. You can think of it like this: ActionScript is the language of Flash Player, and everything in Flex is eventually distilled into ActionScript.

Tags Are Classes

MXML components in a Flex application can be reduced to equivalent ActionScript code. For example, the following code creates a **Button** in MXML:

```
<s:Button id="myButton"/>
```

But you can also add a **Button** using ActionScript:

```
import spark.components.Button;
var myButton:Button = new Button();
addChild(myButton);
```

Knowing this, you can use ActionScript to create components dynamically, without having to rely on MXML.

Attributes Are Properties

When you add attributes to a tag, you're essentially changing the properties of an instance of that component. For example, to change a button's **label** property, you do the following in MXML:

```
<s:Button id="myButton" label="Click Me"/>
```

Behind the Scenes

Want to see all the ActionScript code that Flex generates from MXML? Within your project's properties dialog (Project→Properties), go into the compiler options by choosing Flex Compiler from the list on the left. You should see a field called "Additional compiler arguments". Within this field, add the compiler argument **-keep**, separating it from any other compiler arguments with a space.

This command will save ActionScript code the compiler creates from MXML and will place the files in a folder called *generated*.

which has the equivalent ActionScript:

```
import spark.components.Button;
var myButton:Button = new Button();
myButton.label = "Click Me";
addChild(myButton);
```

Looking deeper at this ActionScript example, you'll see the first line contains an **import** statement, importing **spark.components.Button**. This is the way to tell the compiler you want to bring a certain class into your application.

The second line creates an instance of the **Button** called **myButton**. Following the **Button** type declaration, the statement **new Button()** initializes the class for use.

Unlike framework classes, fundamental data types do not require initializing, meaning you can create them without using the **new** keyword:

```
var name:String = "Bmose";
```

Or for a **Number**, use:

```
var num:Number = 23;
```

NOTE

We'll discuss styles in depth in Chapter 15.

Attributes Are Styles

As you saw in Chapter 3 when we examined the Properties pane, a component can have many types of properties. Under the Category view of the Properties pane, perhaps you noticed a section called Styles. *Styles* are special component properties that are used to define the look and feel of a component.

Although styles are like properties, they are not directly accessible in ActionScript using dot notation. In MXML, you can easily set styles using attributes. However, because of the way styles are implemented in Flex, there is a different syntax for accessing style properties in ActionScript, using the methods **getStyle()** and **setStyle()**.

For example, the **cornerRadius** style property sets the roundness of a **Button** control, and you could set this style with the following MXML:

```
<s:Button id="myButton" cornerRadius="14"/>
```

Because it is a style property, you cannot access it directly. Therefore, the following code is incorrect:

```
myButton.cornerRadius = 14;
```

Instead, you must use the **setStyle()** method. This method takes two parameters. The first is the name of a style property, generally as a **String**, and the second is the value you want to assign it:

```
myButton.setStyle("cornerRadius", 14);
```

The method **getStyle()** allows you to get the current value of a style property. It takes a single parameter, the name of the style you want to access, and returns the value of the style. To get the **cornerRadius** of **myButton**, for

example, you could use the following `ActionScript`, which stores the value in a variable called `roundness`:

```
var roundness:Number = myButton.getStyle("cornerRadius");
```

Attributes Are Event Listeners, Too

Event listeners are triggers the Flex UI components use to identify and respond to various interactions. You created a simple event handler when you built the inline `click` handler for a `Button`. However, while these listeners can be created as MXML attributes, they are not true properties. Let's see why.

Previously, you added a `click` listener to an MXML `Button` declaration like this:

```
<s:Button id="myButton" click="doSomething()" />
```

In `ActionScript`, though, you would set up a click event listener like this:

```
import spark.components.Button;
var myButton:Button = new Button();
myButton.addEventListener("click", doSomething);
addChild(myButton);
```

The `ActionScript` approach uses the `addEventListener()` method. This method takes two parameters: the name of the event—`click` in this case—and the name of a function to call when that event occurs—`doSomething()`.

Unlike inline `ActionScript`, the `addEventListener()` method doesn't allow you to place any expressions (e.g., `textInput.text = "some text"`) directly within the listener declaration. Such activity is permissible only within a true function.

When using the `addEventListener()` method, add only the name of the function you want to call for the second parameter. *Don't include parentheses following the function name.* For example, this code would be incorrect:

```
myButton.addEventListener("click", doSomething());
```

Comments?

You'll find countless reasons to comment code, either to make a note for yourself or for others, or to temporarily prevent code from compiling. Commenting syntax is different between MXML and `ActionScript`.

In `ActionScript`, you can comment either one line or a block of code. To comment out a single line, use the double forward slash (`//`) syntax:

```
//This code won't compile
//public var foo:String = "No Comment";
```

For multiple lines, it's usually more convenient to use comment block syntax:

```
/*
    public var foo:String = "No Comment";
    public var bar:String = "Don't want to see it";
*/
```

NOTE

It's easy to add a comment in Flash Builder, even without knowing the correct syntax. Just highlight the code you want to comment, and then select Source→Toggle Block Comment.

Alternatively, use a keyboard shortcut. In Windows, use Ctrl-Shift-C. For Mac OS, use Command-forward slash (/).

Whether you're editing ActionScript or MXML, the proper comment syntax will be inserted. If you're commenting out code, be sure you select complete MXML tags or script blocks; otherwise, you'll get a compile-time error.

MXML requires a different comment syntax than ActionScript. MXML, which you know derives its syntax from XML, uses the same commenting system as HTML, namely, `<!--` and `-->`. Here's an example of commenting in MXML:

```
<!-- This is a comment in MXML -->
<s:Button label="Button to Keep"/>
<!-- <s:Button label="Button to Remove"/> -->
```

One helpful use of comments is commenting out code to prevent it from compiling. This can be useful when debugging, which we discuss in the next chapter.

Summary

This chapter introduced you to the basics of scripting in ActionScript, and you learned the following:

- ActionScript can be placed inline, within MXML attributes.
- Named functions should be placed within a Script/CDATA block, and they can be called from inline script as well as other functions.
- Named functions can receive parameters and/or return typed values.
- Functions that do not return a value should be typed **void**; functions that do return a value should be strongly typed.
- You can define the scope/accessibility of both variables and functions to control their visibility to other components in an application.
- Classes are specialized, predefined objects that can have both properties (variables) and methods (functions).
- ActionScript classes expect to belong to a package, which you have the option of maintaining outside of any particular project.
- Flex framework components correspond to ActionScript classes.

You covered a lot of ground in this chapter. However, the coolest stuff happening in Flex requires a much deeper understanding of ActionScript, and excelling at ActionScript will take some further study, so we have a few recommendations:

Learning ActionScript 3.0 (O'Reilly)

Foundation ActionScript 3.0 for Flash and Flex (friends of ED/Apress)

Visual Quickstart Guide: ActionScript 3.0 (Peachpit Press)

ActionScript 3.0 Cookbook (O'Reilly)

We'll continue throwing ActionScript at you, and hopefully you'll pick up a few tricks. In the next chapter, we give you a crash course in debugging Flex applications.

DEBUGGING FLEX APPLICATIONS

Butch Cassidy: "I'll jump first."

Sundance Kid: "Nope."

Butch Cassidy: "Then you jump first."

Sundance Kid: "No, I said!"

Butch Cassidy: "What's the matter with you?!"

Sundance Kid: "I can't swim!"

Butch Cassidy: "Why, you crazy...the fall will probably kill ya!"

—Paul Newman and Robert Redford,

Butch Cassidy and the Sundance Kid

IN THIS CHAPTER

Outputting Values to the Console Using `trace()`

Inspecting Event Properties with `trace()`

Using Breakpoints

Summary

By now, you're probably comfortable enough with Flex basics to have created a few experimental interfaces. To be a real code mechanic, though, you also need some techniques to inspect what's happening under the hood. It's called debugging. No doubt you've heard the term before, and your first impressions are probably close enough. Debugging is troubleshooting, and troubleshooting is tedious.

Fortunately Flash Builder includes some tools to simplify your debugging routines, and this chapter teaches you how to use them. We'll discuss use of the `trace()` statement to preview variable values, how to pause and inspect code logic using breakpoints, and how to benefit from Flash Builder's Debugging perspective.

The goal of *debugging* is testing variables and functions to ensure they result in the values you expected when you first created them. By its very name, debugging implies that there's a *bug*—or glitch—in your software that you need to locate. As you'll see, however, there's more to debugging than that.

Our first foray into debugging will expose the `trace()` statement and a new pane from the Flash Builder workbench—the Console pane.

Outputting Values to the Console Using trace()

NOTE

The Console pane is, by default, located at the bottom of your Flash Builder window, next to the Problems pane. If your Console pane isn't visible, make it visible by selecting Window→Console.

The **trace()** statement displays text in the Console pane while you interact with your Flex application. Use of the **trace()** statement requires an event to trigger a function that calls **trace()**, and inside the call statement some application variables are added. The function in Example 6-1 demonstrates proper construction of a **trace()** statement.

Example 6-1. Tracing a control property and a variable value

```
public function someFunction():void{
    trace("myTextInput.text = " + myTextInput.text +
        " and myVariable = " + myVariable);
}
```

To some degree, **trace()** statements are effectively “invite only,” and running an application in the usual manner won't display trace output. To use **trace()**, you'll have to be debugging. So how do you do that? Pretty easily. Instead of launching the application with the green arrow button, click the green bug icon to its right (Figure 6-1).

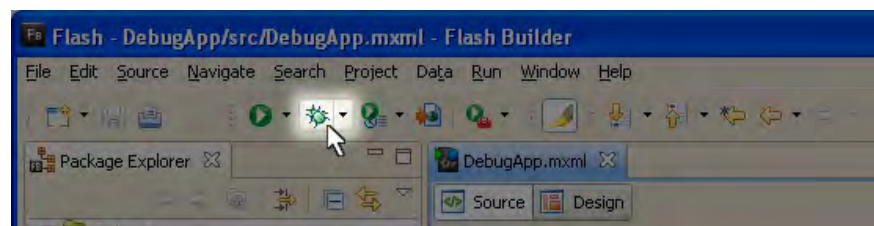


Figure 6-1. Launching Debug mode

Clicking the Debug button launches the application in Debug mode. For the most part it will look the same, but any event calling **someFunction()** will trace its output to the Console pane.

So let's demonstrate how to use **trace()**. For that we'll need a new Flex project. Assuming you're already in Flash Builder, open the File menu and select New→Flex Project. Title the application **DebugApp** and save it under C:\Flex4Code\DebugApp. Accept the rest of the default settings, and when the editor opens, piece together the application as shown in Example 6-2.

Example 6-2. Application code we'll use to demonstrate debugging

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx" >
```

```

<fx:Script>
  <![CDATA[

    private var numericStepperValue:int;

    private function onChange():void{
      //do something
    }
    private function doAdd():void{
      //do something
    }
    private function doMultiply():void{
      //do something
    }

  ]]>
</fx:Script>

<s:VGroup gap="25" verticalCenter="0" horizontalCenter="0">

  <s:HGroup verticalAlign="bottom">
    <s:Label text="Input (Select Number):"/>
    <s:NumericStepper id="myNS" minimum="1" maximum="100"
      change="onChange()"/>
  </s:HGroup>

  <s:HGroup verticalAlign="bottom">
    <s:Label text="Output:"/>
    <s:TextInput id="myTI" text="My number is: {myNS.value}"/>
  </s:HGroup>

</s:VGroup>

</s:Application>

```

You can run it as soon as it's built, and it's pretty simple. The MXML tags provide input and output controls, a **NumericStepper** (**myNS**) and a **TextInput** (**myTI**), respectively. The functions under the script tag are hollow, and as you can tell by the comments (*//do something*), this is where we'll conduct our **trace()** experiments.

This piece in the **TextInput** control's **text** attribute—**{myNS.value}**—binds the current value of the **NumericStepper** control to the binding (**{}**) position within the **TextInput** control's **text** attribute. This syntax is called data binding, which we've seen a couple of times already, and which we discuss in depth in Chapter 8.

Presently, the **NumericStepper** is calling the empty **onChange()** function, and nothing is happening when it's called. We'll create a **trace()** there. So go ahead and delete the comment, and rework the function to look like Example 6-3.

Example 6-3. Adding a `trace()` statement to the `onChange()` function

```
private function onChange():void{
    numericStepperValue = myNS.value;
    trace("testing stepper value: " + numericStepperValue);
    doAdd();
}
```

This block first assigns our variable, `numericStepperValue`, the current value of the `NumericStepper` control. Next, the `trace()` statement outputs our variable to the Console pane. The last line, `doAdd()`, sets up the next function. For now it's OK that `doAdd()` is empty.

With everything in place, click the Debug icon and watch your Console pane as you ratchet up the `NumericStepper` (Figure 6-2).

WARNING

If you don't have a multimonitor display, you might need to resize your browser window in order to watch the Console pane while you increment the `NumericStepper`. See Figure 6-2 for an example of this.

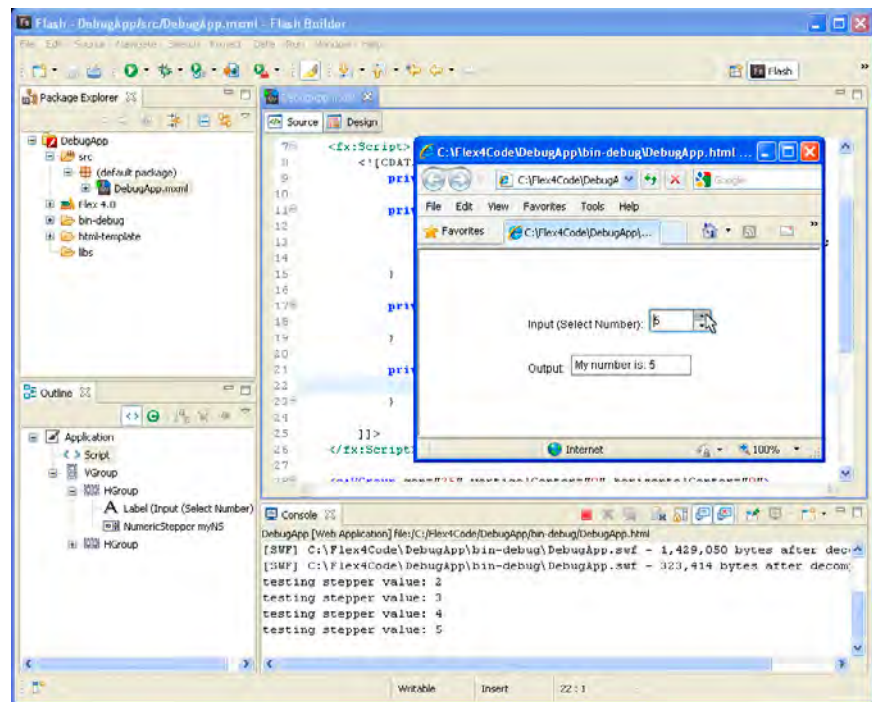


Figure 6-2. Tracing variable values as output to the Console pane

That's simple enough. But let's complete the other functions to see some more action. Modify the `doAdd()` and `doMultiply` functions accordingly, as shown in Example 6-4.

Example 6-4. Adding more trace() statements

```
private function doAdd():void{
    numericStepperValue += 10;
    trace("adding 10 to stepper: " + numericStepperValue);
    doMultiply();
}
private function doMultiply():void{
    numericStepperValue *= 10;
    trace("multiplying stepper by 10: " + numericStepperValue);
}
```

Now when you run Debug mode, notice that your application will continue to output the incrementing value of the **NumericStepper** control, but the Console pane will report the additional math carried out on our variable.

As you have seen, the **trace()** statement is easy to work with and useful for monitoring application variables. Next you'll learn how to use **trace()** and Debug mode to inspect information about an event.

Inspecting Event Properties with trace()

Last chapter we learned that you can pass parameters into functions. To explore another use of the **trace()** statement, we're going to do just that. For this exercise we need to add a new function, **checkEvent()**, and modify the **NumericStepper** control's **change** attribute to call that function.

Add the **checkEvent()** function directly below **doMultiply()**. The code is shown in Example 6-5.

Example 6-5. Tracing an event's type and the object that called it

```
private function checkEvent(event:Event):void{
    trace("The event type is: " + event.type +
        " and the current target is: " + event.currentTarget);
}
```

Notice the code we added between parentheses, **event:Event**. This line tells our function to expect a variable of type **Event** and name it **event**. You can call it whatever you like, but **event** in lowercase does the job.

Now we have a **trace()** statement capable of reporting details about an event, namely the event's **type** and **currentTarget**. Did you realize using this function will require one more step? You still have to call the function and pass it a parameter.

The **event** parameter is somewhat special. It's built in, meaning you don't have to create it before you can use it; rather, Flex will create it at compile time. Think of it as a feature. That said, you can pass the **event** parameter into the **checkEvent()** function by modifying the **change** attribute of the **NumericStepper** like in Example 6-6.

Two Heads Are Better Than One

If you've ever wanted an excuse to get a secondary monitor (or even a third), serious debugging may provide the impetus. Of course, multiple displays are more expensive, and sometimes they can be tricky to set up. But be warned that once you become accustomed to developing with a multiple monitor setup, it's difficult to go back.

NOTE

If you peruse enough code samples on the Web, you'll often see **Event** variables abbreviated as simply **e**, or sometimes **evt**.

Example 6-6. Calling the event tracer using inline script

```
<s:NumericStepper id="myNS" minimum="1" maximum="100"
  change="checkEvent(event)"/>
```

Here's a breakdown of what happens when you launch the application in Debug mode:

1. When you increment the **NumericStepper**, its **change** method calls **checkEvent()**.
2. The **checkEvent()** function will receive an object of type **Event**, which is a Flex class possessing unique properties.
3. The **event** object's **type** property will be inspected and output to the Console pane. In this case, it is a **change** event.
4. The **event** object's **currentTarget** property, which contains a reference to the object that passed in the event (whatever called the function), will also be discovered and reported. And in this case, it is the **NumericStepper (myNS)**.

Because the function expects an **event** parameter to be passed in, failing to pass it when calling the function will incur a compile-time error.

You're ready to test this, so launch the application in Debug mode and increment the **NumericStepper**. You should get a message in the Console pane that states:

"The event type is: change and the current target is DebugApp.ApplicationSkin2._ApplicationSkin_Group1.contentGroup.VGroup5.HGroup6.myNS"

Make sense? You knew the type of the event would be **change**, but the **currentTarget** value is a little cryptic. First, look at the end of the phrase and note **myNS**. Remember, **myNS** is the **id** of our **NumericStepper**. But what's the rest?

When Flex compiles your application, it creates a class based on the name of the application—**DebugApp** in this case—and the **currentTarget** property describes the fully-qualified containment order from the lowest container—**DebugApp**—all the way to the control that called our function—**NumericStepper**. So, in terms of hierarchy, **myNS** is considered part of an **HGroup**, which is in turn part of a **VGroup**, and so on.

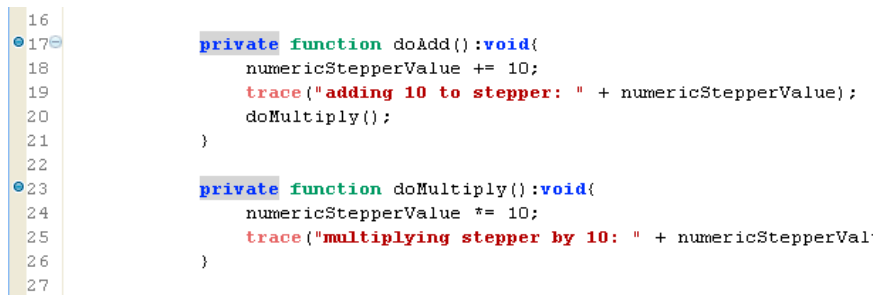
This event information can be handy when creating functions because you can make the functions highly reusable. For example, you could create a function that first tests for the type of event that occurred and/or what called it (the **currentTarget**), and then react accordingly.

Using Breakpoints

In this section we discuss how to use breakpoints to pause your application while you examine control properties and variable values.

A *breakpoint* is a position in the logic of your code where you'd like to stop everything and conduct an inspection. Some programming constructs can get quite complicated, so it's useful to say, "Halt!" and then review the current status of one or many variables. Breakpoints allow you to do just that—pause and take a look.

To create a breakpoint, simply double-click a line number in your code editor where you want to suspend your application. To test this, we'll create two breakpoints, one inline with `doAdd()` and the other inline with `doMultiply()`; see Figure 6-3. We'll also need to restore the `change` method of our `NumericStepper` to call `onChange()` once again, shown in Example 6-7.



```

16
17 private function doAdd():void{
18     numericStepperValue += 10;
19     trace("adding 10 to stepper: " + numericStepperValue);
20     doMultiply();
21 }
22
23 private function doMultiply():void{
24     numericStepperValue *= 10;
25     trace("multiplying stepper by 10: " + numericStepperVal
26 }
27

```

Figure 6-3. Breakpoints created to interrupt the two math functions in `DebugApp`

Example 6-7. Restoring the `onChange()` function to test breakpoints

```

<s:NumericStepper id="myNS" minimum="1" maximum="100"
    change="onChange()"/>

```

At this point, you're ready to launch the application in Debug mode. Whenever the `NumericStepper` is incremented, `onChange()` will be called, and everything will abruptly stop before `doAdd()` is called. Let's see how it works. Go ahead and launch Debug mode.

Everything will seem normal at first. Your application will load in a browser window and wait for you to interact with it. Once you increment the `NumericStepper`, however, you'll trigger the `onChange()` function as well as the first breakpoint.

You'll probably be prompted with a dialog box from Flash Builder similar to Figure 6-4, asking if you want to switch to the Flash Debug perspective. You might want to enable "Remember my decision" so Flash Builder will launch the correct perspective whenever breakpoints are set. Either way, click "Yes" and you'll gain a whole new perspective on debugging.

NOTE

Breakpoints will stop your application before any code on that line is processed. Therefore, it's best to place breakpoints on the same line calling a function so you can inspect variable values both before and after they are changed by a problematic routine.

NOTE

To hit the first breakpoint in `DebugApp`, you'll need to increment the `NumericStepper`, which will call the code corresponding to the breakpoint.

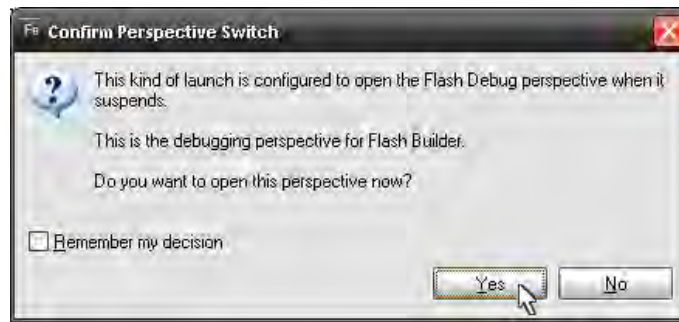


Figure 6-4. Flash Builder's Confirm Perspective Switch dialog box

Seeing a New Perspective

We briefly mentioned perspectives in Chapter 2, but in this section we demonstrate how a different perspective can benefit you as a developer.

Perspectives in Flash Builder are preset workbench layouts that include the most convenient panels and panes for a particular job. Before switching to the Debug perspective, you were using the Flash perspective. The Flash perspective is geared toward writing and editing code; among other components, it contains the Package Explorer, the Outline pane, and the Properties pane. When debugging, you'll need some panes we haven't discussed, and they're contained in the Debug perspective.

The Variables Pane

One of the new panes you'll see in the Debug perspective is the Variables pane, and by default it's in the top-right panel.

The Variables pane presents information in a tree list that needs a lot of vertical space, so you might want to change its positioning. Flash Builder allows you to move and resize panels easily. Just click and drag the Variables tab toward the right until it snaps against the right side of Flash Builder, becoming its own panel. It may also help to widen it. Alternatively, if you have multiple monitors, you can detach the pane and place it somewhere more convenient.

The Variables pane displays information about your application's current state when the debug runtime hits a breakpoint in your code. You'll have two columns of information at your disposal: Name and Value. Name identifies components, variables, etc., and Value identifies their current value.

You should see a menu item called **this** in the tree list. Note that **this** corresponds to the **DebugApp** application.

Expand the contents of **this** by clicking the plus symbol (+), and then look for the items **myNS**, **myTI**, and **numericStepperValue**. The child nodes **myNS** and **myTI** refer to our controls, and **numericStepperValue** refers to our **int** variable (Figure 6-5).

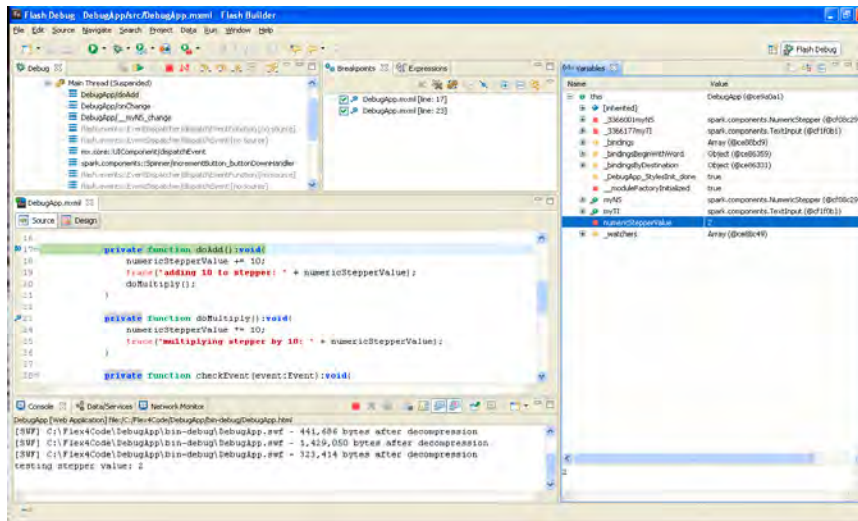


Figure 6-5. Reviewing **numericStepperValue** in the Variables pane.

To get to the next breakpoint, we have two options: we can either walk through the code line by line, or we can leap to the next breakpoint using the Resume button. First, let's go for a walk.

“Walk, Don't Run”

Under your Run menu and along the top of your Debug pane, you'll find a few tools to dictate how you want to proceed while in Debug mode. You'll find these functions quite handy, and you'll appreciate them even more if you memorize their corresponding keyboard shortcuts. Let's consider the debug-
ging controls, shown in Figure 6-6.

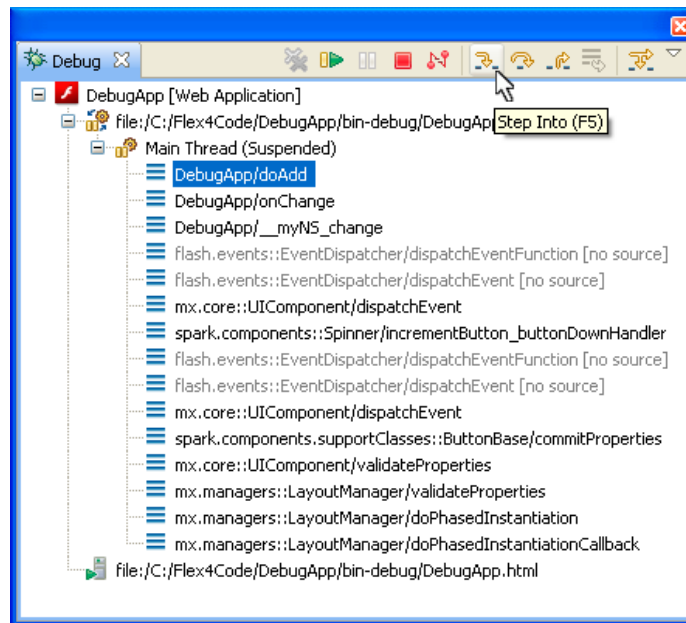


Figure 6-6. Debugging controls in the Debug pane

NOTE

If the stepping shortcuts are not the same for Mac OS, you can open the Run menu and glean the correct keyboard shortcut from the menu.

Step Into (F5 in Windows)

Step into a called function and stop at the first line of that function.

Step Over (F6 in Windows)

Execute the current line of a function, and then stop at the next line.

Step Return (F7 in Windows)

Continue execution until the current function returns to its caller.

Resume (F8 in Windows)

Resume application processes, either stopping at the next breakpoint or running through to completion.

For our purposes, use the Step Over command (F6) to walk through each individual line of code. Here's a handy trick: if you hover your mouse cursor over a variable, such as `numericStepperValue`, Flash Builder will offer you a quick pop up noting the variable's current value (Figure 6-7).

If you prefer a faster way to watch, find your variable in the Variables pane and watch its value change as you step through the code line by line. When a variable value changes, that variable will become highlighted in the Variables pane. Note how this looks in Figure 6-8.

```

10 private function onChange():void{
11     numericStepperValue = myNS.value;
12     trace("testing stepper value: " + numericStepperValue);
13     doAdd();
14 }
15 private function doAdd():void{
16     numericStepperValue += 10;
17     trace("adding 10 to stepper: " + numericStepperValue);
18     doMultiply();
19 }
20 private function doMultiply():void{
21     numericStepperValue *= 10;
22     trace("multiplying stepper by 10: " + numericStepperValue);
23 }

```

Figure 6-7. Spot-checking a variable's value with a quick mouseover

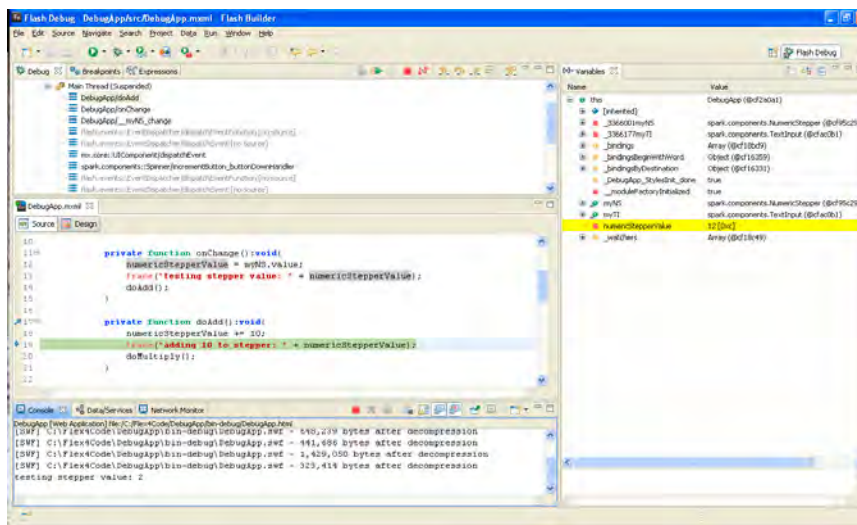


Figure 6-8. Watching a variable's value change in the Variables pane

We previously mentioned that another debugging option is to leap directly to the next breakpoint or to the end of the processing sequence. Once you're content with your ability to follow a variable through the runtime using the stepping approach, and assuming you've exhausted your logical sequence, increment the **NumericStepper** once again to trap the runtime at the first breakpoint. Then, either toggle the Resume button, which is in the top-left panel of your Debug perspective (Figure 6-9), or hit F8 (Windows) to leap to the next breakpoint. Resuming a third time should exhaust the process, placing you once again at the end of the sequence.

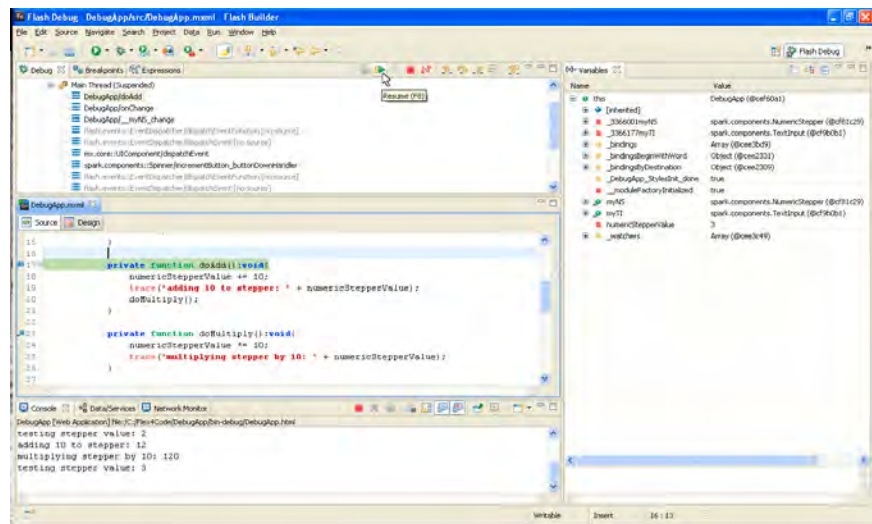


Figure 6-9. Advancing to the next breakpoint using the Resume button

Ending Your Debug Session

Once you're ready to quit debugging, be sure to end your session using the red Terminate button located to the right of the Resume button (Figure 6-10). It's also available in the Console pane, or if you're a Windows user, via the keyboard shortcut Ctrl+F2. Because debugging pauses Flash Player, you might run into problems if you try to close the browser without properly ending the debug session.

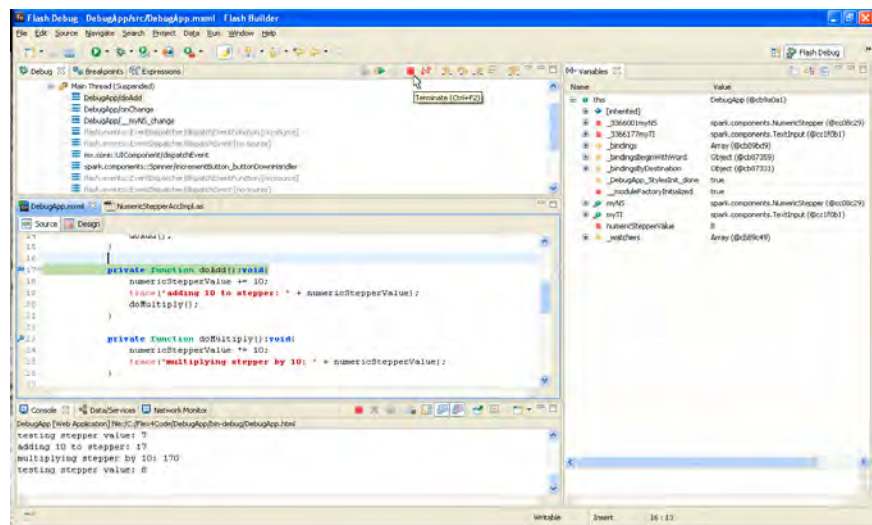


Figure 6-10. Ending the debug session using the Terminate button

Finally, return to the basic Flash perspective using the toggle button at the top right of Flash Builder or by selecting Window→Perspective→Flash (Figure 6-11).

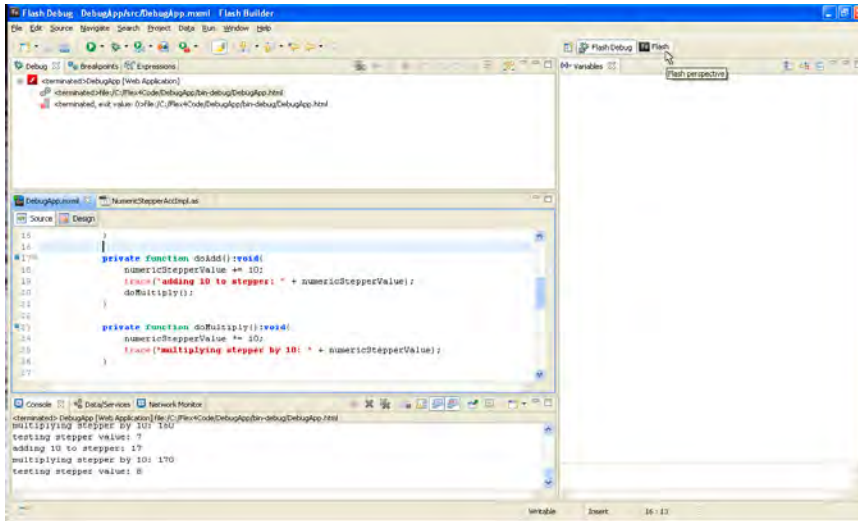


Figure 6-11. Returning to the Flash perspective

Summary

Except for the occasional mind-bending troubleshootings, debugging isn't that bad. And hopefully you'll find that typical day-to-day debugging scenarios can be stimulating and a little fun. After all, if programming were 100% easy, you wouldn't experience that thrill when your applications work as intended.

This chapter introduced you to three debugging techniques:

- Using a **trace()** statement to reveal component properties and variable values
- Using breakpoints to trap your application in the runtime while you snoop
- Walking through your code line by line while watching for value changes

In Chapter 7, we continue working with ActionScript as we examine event handling a little more closely—and make a fun game, no less.

ADDING INTERACTION WITH ACTIONSCRIPT

Software users excel at initiating interaction. Their techniques include mouse movement, clicking, dragging and dropping, and keyboard entry, to name a few.

For you, the Flex developer, “the trick” isn’t programming each individual event interaction as much as it is “seeing the big picture,” or specifically, knowing what interactions and events the framework exposes and the relatively few techniques used to weave them into your application.

In this chapter, we review the most common events; we also teach you how to research events in the official documentation. After we finish discussing the background information, we demonstrate how to handle events in your programs, and for the grand finale, we develop a simple, yet surprisingly fun game called Collision! (Figures 7-1 and 7-2), which we hope will inspire you through the rest of this book.

IN THIS CHAPTER

- Understanding Events
- Common Events
- Researching Events
- Listening for and Responding to Events
- Collision! A Whirlwind of Events
- Summary

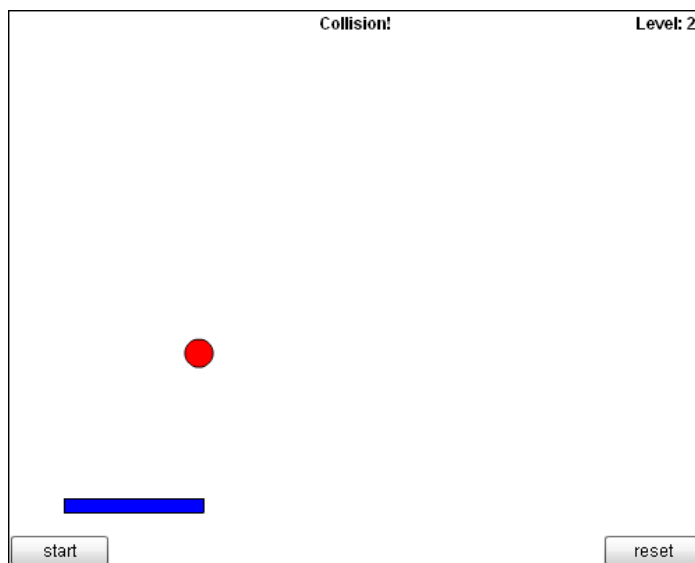


Figure 7-1. Collision!: 3 files, 190 lines of code, 1 timeless classic

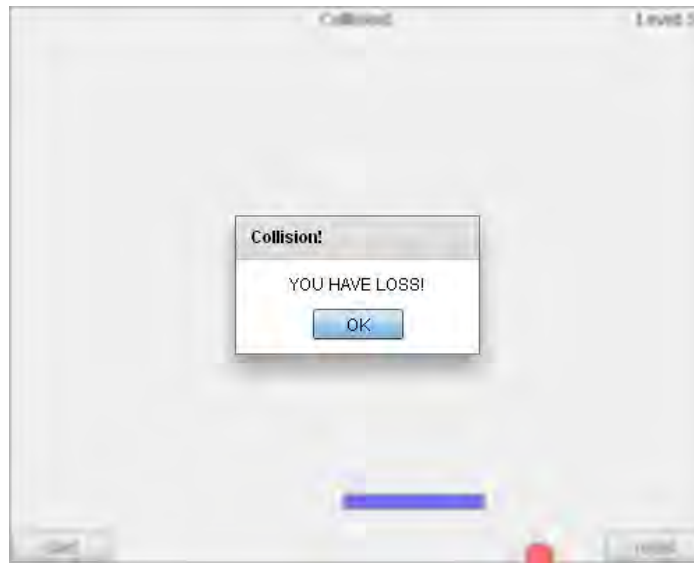


Figure 7-2. Keeping it real with grammar from the Nintendo days

Understanding Events

Flex applications listen for and respond to *events*. An event is something that happens either by direct user interaction or by something less obvious, such as a component being drawn onscreen or data returning from a server. When a user clicks a **Button**, a **click** event occurs. When a **TextInput** field receives or loses a character, a **change** event occurs. When the application display finishes rendering, the **applicationComplete** event occurs.

NOTE

Interaction and event handling can extend beyond the user interface. For instance, if your application makes a request to a web server, that server's response is an event. We experiment with interactions like these in Chapter 11, where we'll build a searching utility, and Chapter 16, where we'll integrate with PHP and MySQL.

Programming events in Flex applications requires code to both *listen for* as well as *respond to* or *handle* events. When an event occurs, you say the event *fired* or was *triggered*, or even *dispatched*. For example, clicking on an image thumbnail may fire a **click** event that loads the full-size image, but the loading in turn triggers two more events—**progress** and **complete**. As such, the **progress** event can be listened for and handled by a **ProgressBar**, and the **complete** event can be used to hide the **ProgressBar**.

Common Events

Programming for interaction and events requires knowing the options available to you. Table 7-1 lists the most common events, their constant values, and a description of the event. We discuss constants later in this chapter, in the section titled “Handling Events in ActionScript” on page 113.

Spend a moment reviewing the events described in this table. Although you might not have an immediate use in mind for each event, familiarize yourself with this list now so that you're better prepared later when you really need these events.

Table 7-1. *The most common events*

Event name	Constant	Description
change	Event.CHANGE	Fired when a selection changes in a list or navigation component such as a TabBar , or when a text component's text is changed.
click	MouseEvent.CLICK	Fired when a user clicks an element. This means someone pressed the mouse button down <i>and</i> released it on the same component.
mouseDown	MouseEvent.MOUSE_DOWN	Fired when someone presses the mouse button down on a component.
mouseUp	MouseEvent.MOUSE_UP	Fired when someone releases the mouse button on a component.
rollOver	MouseEvent.ROLL_OVER	Fired when the mouse pointer moves over a component.
rollOut	MouseEvent.ROLL_OUT	Fired when the mouse pointer moves out of a component's area.
initialize	FlexEvent.INITIALIZE	Fired during the draw process when a Flex component or the application container is being instantiated.
creationComplete	FlexEvent.CREATION_COMPLETE	Fired when a Flex component is finished drawing.
resize	Event.RESIZE	Fired when the application is resized because the browser or window is resized.
applicationComplete	FlexEvent.APPLICATION_COMPLETE	Fired when the application is fully initialized and drawn in the display.

Researching Events

We realize a list of common events will only get you so far. Therefore, it will benefit you to know the best resources for researching events on the fly.

What Events Are Available to This Component?

You won't always remember the full range of events available to a component. When this is the case, there are a few approaches to help you find out fast:

- In Design mode, select a component in the application, and then check the Properties pane in Category view. Find the Events section, and you'll have a list of every event available for that component (Figure 7-3).

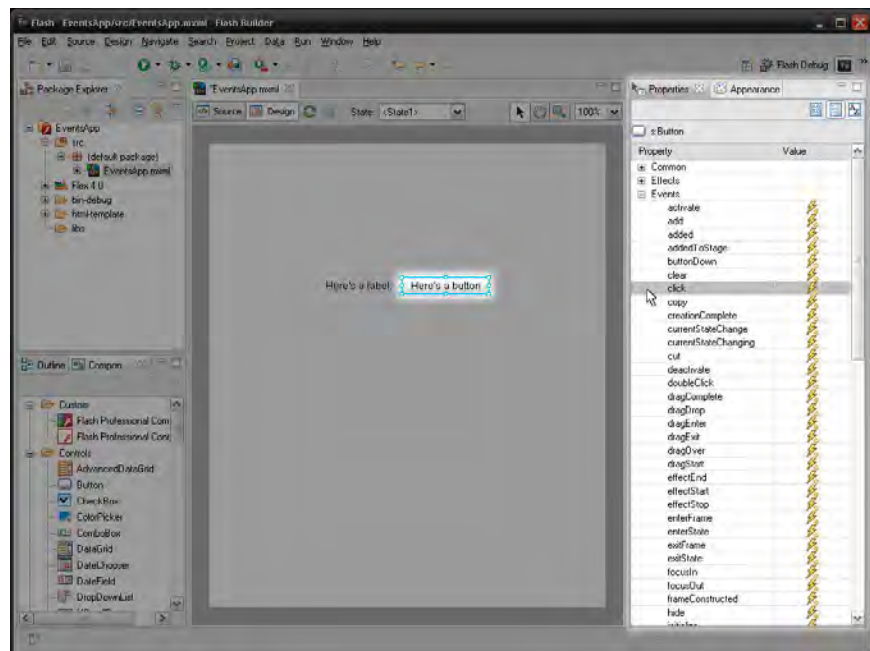


Figure 7-3. Identifying events using the Properties pane

- In Source mode, code completion will reveal every property available to a component, including events. Events are indicated by a lightning bolt icon (Figure 7-4).

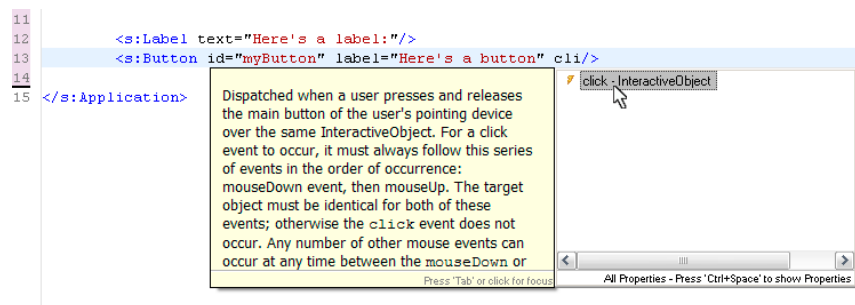


Figure 7-4. Identifying events using the code completion hints

- Review the official documentation for a component.

Flex/ActionScript Documentation

Sometimes you'll need to know more than “what events are available to this component?” You might be left asking, “When can I use this, and how can I use it?”

When this is the case, there are three approaches you can take to researching event specifics: Flash Builder's help system, ASDoc, and the favorite, Adobe's Flash Platform Language Reference.

Flash Builder's help system

To access Flash Builder's help system, follow Help→Flash Builder Help.

You'll be presented with a utility that combines internal and external documentation. If you use the search utility to search for a component, technique, or tutorial, the help system will expose a number of articles, including official documentation and contributions from the user community, describing anything and everything related to your search.

ASDoc

If you're in the middle of coding and you just want a quick review of a component, open the ASDoc pane (Window→ASDoc), select a component, and then read through the information provided (Figure 7-5).

This sort of documentation did not exist in Flash Builder's predecessor, and it can be quite useful, especially for reviewing information at a glance. However, ASDoc won't give you in-depth coverage or example code; for that, you'll need to check Adobe's Flash Platform Language Reference.

NOTE

You need to be in Source mode to open the ASDoc pane.

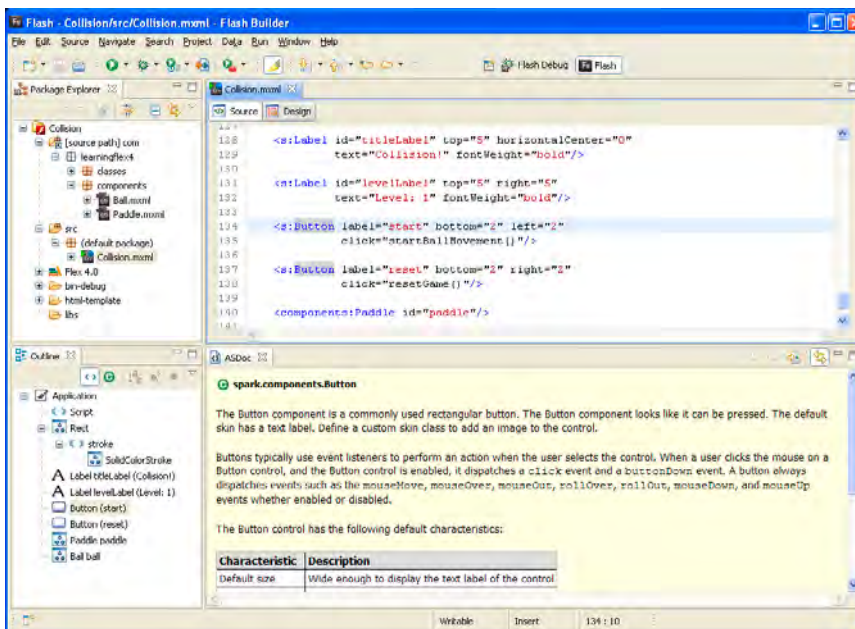


Figure 7-5. Reviewing component details at a glance in the ASDoc pane

NOTE

Adobe's Language Reference has gone through a recent rebranding of sorts. With Flex 3, we would have discussed the Flex 3 Language Reference, and the same applied to Flex 2. However, with Flex 4, Adobe is emphasizing the Flash ecosystem, and as such, the Language Reference for each of the participant technologies has been amalgamated into one universal Flash Platform Language Reference.

Adobe's Flash Platform Language Reference

When you need to deeply research a Flex component, turn to Adobe's Flash Platform Language Reference:

<http://livedocs.adobe.com/flex/gumbo/langref/>

The Language Reference is dense, and arguably the most comprehensive resource available to Flex/ActionScript developers. Not only does the Language Reference describe just about every aspect of a component or class (e.g., properties, methods, events, styles, etc.), it also provides examples, and better yet, it contains a “footer” section of questions and answers, comments, and insights contributed by both the user community and Adobe moderators.

If there is a downside to the Language Reference, it's that it can be daunting at first, particularly for new developers. But once you're comfortable with it, you'll begin to appreciate its depth of coverage.

As you research topics in the Language Reference, you'll see something similar to Figure 7-6. At the top of every component's reference is a list of its properties, methods, events, styles, and so on.

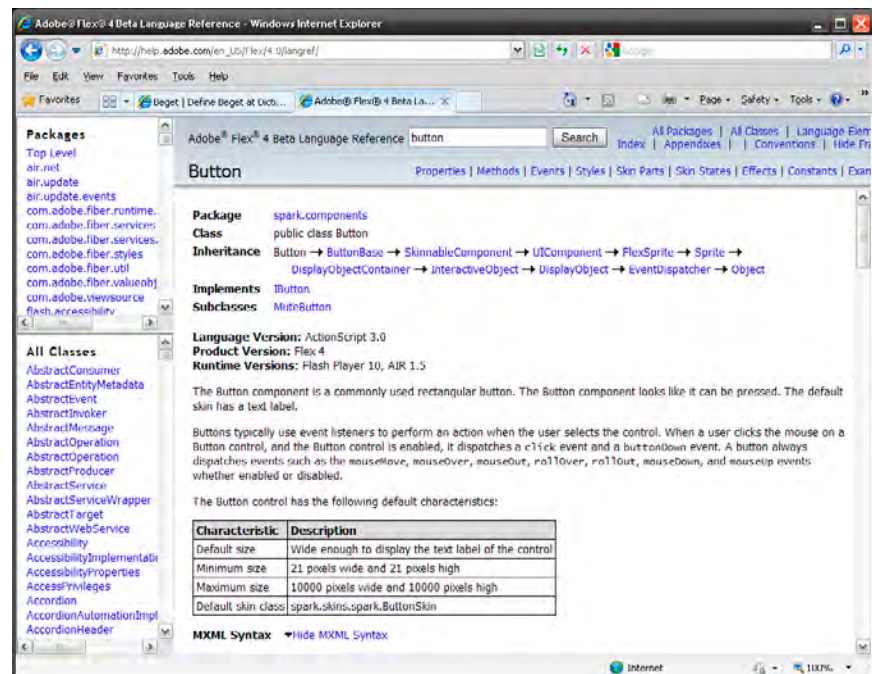


Figure 7-6. Adobe's Flex 4 Language Reference

To inspect the events supported by a component, click the Events link at the top, or scroll down to the Events heading. There you'll find the list of events as pictured in Figure 7-7.

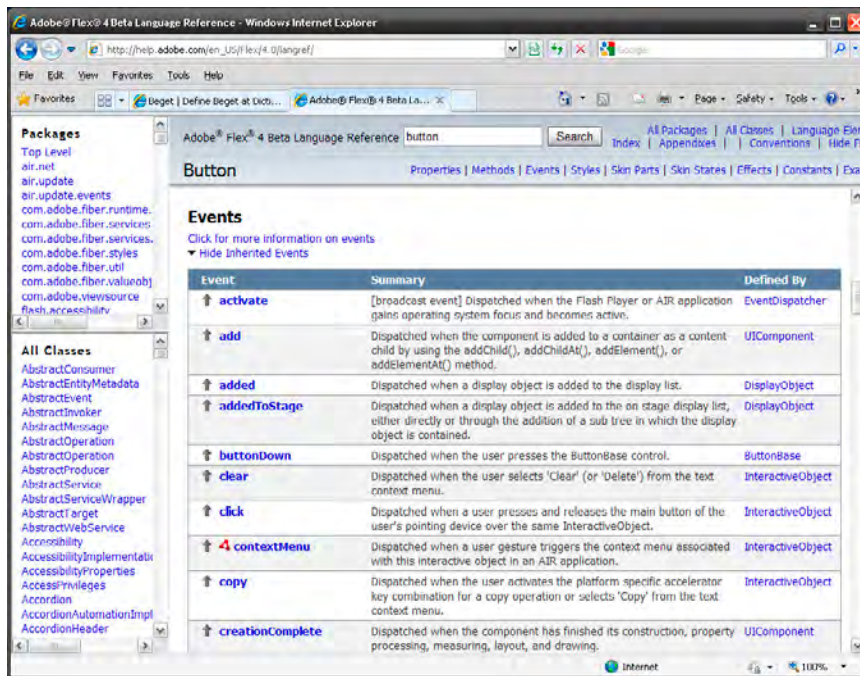


Figure 7-7. The Flex Language Reference detailing a Button's events

Note that there may be many inherited events you won't see unless you select "Show Inherited Events." Components typically inherit functionality from a more abstract, lower-level class, so to view the properties of the base classes, you'll need to expand the inherited events. The same reasoning applies to properties, styles, and everything else. In this case, the **Button** inherits its **click** event from **UIComponent**, which is the base class for all Flex UI components.

Listening for and Responding to Events

We know you're itching to see some code, and don't worry, you're about to get a sincere dose. Now that you know how to expose and research events, let's consider how you can integrate event listeners and event handlers into your applications. We discuss two approaches:

- Handling and/or calling event handlers using inline script
- Adding event listeners and event handlers within the Script/CDATA block

Handling Events Inline

By now you should be comfortable with the idea of handling events with inline ActionScript. After all, we used this technique in both Chapters 5 and 6.

In Chapter 5 we used inline event handling (**Button**, **click**) to change the **text** property of a **TextInput** control. We also used inline event handling (**Button**, **click**) to call a named function, **clearSpeakTI()**.

Chapter 6, although shorter and more specialized, demonstrated the use of an inline event (**NumericStepper**, **change**) to call the named functions **onChange()** and **checkEvent()**.

These approaches don't change. Script can be placed inside MXML attributes to handle simple, one- or two-line tasks, or to call named functions in the Script/CDATA block to handle more complex, several-line tasks.

Before we move on, let's see two more examples using inline handling. The first example uses pure inline handling to follow a **mouseMove** event and dispatch the mouse's **x** and **y** coordinates to two **TextInput** controls. The second example listens for a **TextInput** control's **change** event to validate an email address.

Inspecting screen coordinates with **mouseMove()**

For Example 7-1 we created a new project, **CoordCapture**, that uses the **Application** container's **mouseMove** listener to send the current application-level mouse coordinates (**mouseX**, **mouseY**) to two **TextInput** controls.

There shouldn't be any surprises here. Do notice, though, that the mouse coordinates return **Number** values, and since the **TextInput** controls are expecting **String** types, we're casting the coordinate properties into **Strings** using the expressions **String(mouseX)** and **String(mouseY)**, respectively.

Example 7-1. *C:\Flex4Code\CoordCapture\src\CoordCapture.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  mouseMove="
    mouseXTI.text=String(mouseX);
    mouseYTI.text=String(mouseY);">

  <s:VGroup verticalCenter="0"
    horizontalCenter="0">

    <s:HGroup verticalAlign="middle">
      <s:Label text="Mouse X: "/>
      <s:TextInput id="mouseXTI"/>
    </s:HGroup>

    <s:HGroup verticalAlign="middle">
      <s:Label text="Mouse Y: "/>
      <s:TextInput id="mouseYTI"/>
    </s:HGroup>

  </s:VGroup>
</s:Application>
```


If you create this application and run it, you'll discover something important about coordinate positioning in Flash—namely, the *origin* (0, 0) exists in the top-left corner of the layout, with *x* increasing as you move right and *y* increasing as you move down. The maximum coordinate values exist at the bottom-right corner. For a demonstration of this, see Figures 7-8 through 7-11. This concept will be particularly important when we create the game at the end of this chapter.

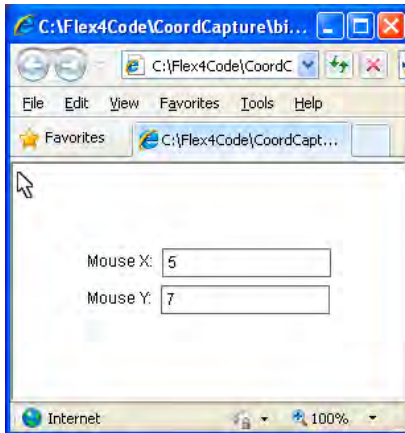


Figure 7-8. Mouse near the origin (0, 0), top left

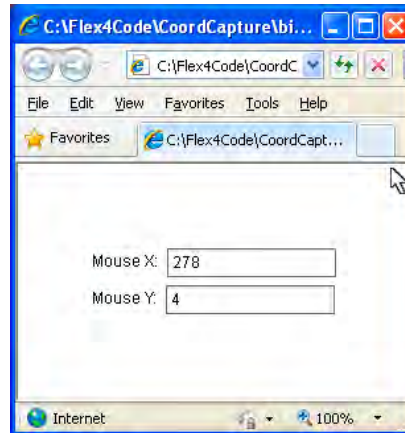


Figure 7-9. Mouse near max-x and min-y, top right

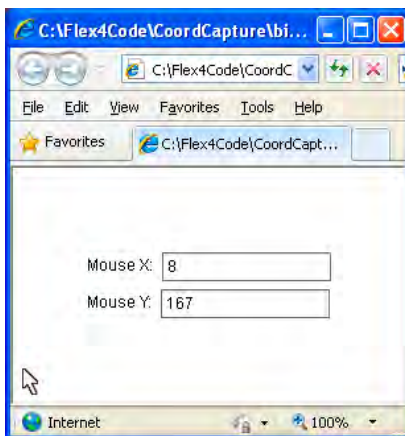


Figure 7-10. Mouse near min-x and max-y, bottom left

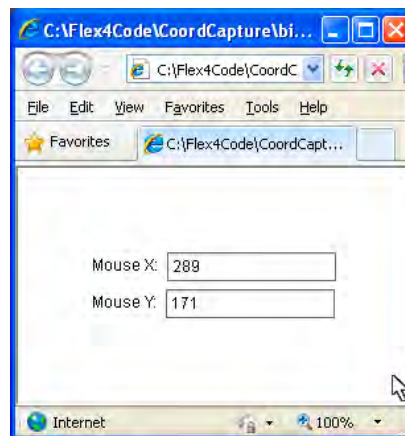


Figure 7-11. Mouse near max-x and max-y, bottom right

Following the mouse programmatically is useful to a variety of applications; however, you also have access to the `mouseover` and `rollover` events, which are available to any class inheriting from `UIComponent`, and which you can use to create so-called “invitations” and “interesting moments” that inspire users to interact with a particular component.

NOTE

Mouse coordinate capture occurs relative to the component doing the listening, but the `MouseEvent` class contains built-in methods to get the coordinate relative to the component (`event.localX` and `event.localY`) or the coordinate relative to the application (`event.stageX` and `event.stageY`). For more details on how to access mouse coordinates in these capacities, check out the Language Reference documentation for the `MouseEvent` class.

NOTE

For a solid discussion of both what to do (best practices) and what not to do (anti-patterns) when creating user interfaces, check out Bill Scott and Theresa Neil's *Designing Web Interfaces* (<http://oreilly.com/catalog/9780596516253/>).

X and Why? A Note About Coordinates

You're probably familiar with the Cartesian coordinate system. With **x** referring to placement along a horizontal axis and **y** referring to placement along a vertical axis, the **origin** is 0,0, and **x** and **y** values increase as you move up or to the right, respectively. In this system, a point at **x**="3" and **y**="5" will be placed up and to the right of the origin (Figure 7-12).

In Flash, the origin is at the top left of the stage (i.e., the parent **Application** container), and **x** and **y** values increase as you move toward the bottom and right, respectively. So, in comparison with what you may be familiar with, the **y** values are backward in Flash.

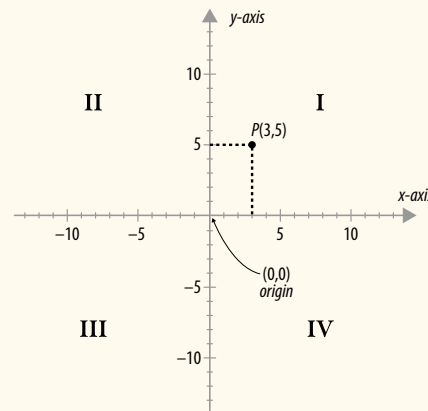


Figure 7-12. A Cartesian coordinate system (image courtesy of Gustavb and Wikipedia, reprinted under the terms of the GNU Free Documentation License)

Testing data input using change()

For Example 7-2 we created the project **ValidateEmail** to demonstrate a practical use of the **change** event to validate one **TextInput** control's **text** property against that of another **TextInput**.

Unlike the previous example, where the entire event is handled inline, this example calls a named function to perform a conditional analysis using an **If..Else** block. The **If..Else** block in this example uses an equality operator (**==**) to test whether both **TextInput** controls contain the same **text** value. Notice that the entire conditional expression—**emailTI.text == validEmailTI.text**—is wrapped in parentheses. If the values on both sides of the expression match, the submission **Button** is enabled and a "VALID!" alert becomes visible; however, if the values do not match, the **Button** is disabled (**submitButton.enable = false**) and the "VALID!" alert is made invisible (**validLabel.visible = false**). See Figures 7-13 and 7-14.

When you create this example for yourself, pay particular attention to the **If..Else** conditional and the construction of the conditional block itself. The ability to make decisions with code is crucial, and this is the first time we've exposed you to something like this. However, we won't let you down; we have some more complex stuff lined up before the end of this chapter.

Confirm Your Email Address

Figure 7-13. Reentering an email address; the submit button remains disabled

Confirm Your Email Address

 VALID!

Figure 7-14. When both fields match, the submit button enables and the "VALID!" indicator appears

Example 7-2. C:\Flex4Code\ValidateEmail\src\ValidateEmail.mxml

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Script>
    <![CDATA[
      private function checkEmail():void{
        if (emailTI.text == validEmailTI.text){
          validLabel.visible = true;
          submitButton.enabled = true;
        }else{
          validLabel.visible = false;
          submitButton.enabled = false;
        }
      }
    ]]>
  </fx:Script>

  <s:VGroup horizontalCenter="0"
    verticalCenter="0">
    <s:Label text="Confirm Your Email Address"
      fontWeight="bold"/>
    <s:TextInput id="emailTI" width="250"
      change="checkEmail()"/>
    <s:HGroup verticalAlign="middle">
      <s:TextInput id="validEmailTI" width="250"
        change="checkEmail()"/>
      <s:Label id="validLabel" text="VALID!"
        color="#267F22" visible="false"/>
    </s:HGroup>
    <s:Button id="submitButton" label="Submit"
      enabled="false"/>
  </s:VGroup>

</s:Application>

```

NOTE

The UI layouts for some of these examples are gradually increasing in complexity, and we admit that we're relying on your ability to make sense of the MXML when it's presented in context. Chapter 9 will address layout construction in greater detail. For the meantime, we promise not to throw anything unreasonable at you.

Handling Events in ActionScript

Up until now all of our examples have demonstrated inline event sequences, with the inline listener either performing some task right in the MXML or calling a named function in a Script/CDATA block. However, you can also set up event listeners using ActionScript, and honestly, this approach affords you some extra liberties—for one, timing. Before we see a couple examples of ActionScript-only event handling, let's discuss something called a *constant* and the *event constants*.

Event constants

A *constant* is a fixed value, which means it can store a value that you don't expect to change. Constants are often used to give a name to a numeric value because, let's face it, words are easier to remember than numbers. Can you remember π (pi) to the 15th decimal place? We didn't think so. What about the ASCII value for the Caps Lock key? That's why some smart person invented constants. Get a piece of the pi by using the constant **Math.PI**, or get the keyboard code for the Caps Lock key with **Keyboard.CAPS_LOCK**.

Calling numeric values is not the only thing constants are good for. The Flex framework often uses constants for **String** values as well, and the most common examples of this are the event types. For instance, the event describing when the mouse goes outside of an application's bounds is called **mouseLeave**. Because the string "mouse leave" or "mouse-leave" isn't sufficient to reference this event, ActionScript establishes event constants so you don't have to remember the specifics: **Event.MOUSE_LEAVE**.

You might be thinking to yourself, "Is that easier to remember?!" Maybe not, but here's the take-home message: constants are properties of a class (**MOUSE_LEAVE** is a property of the **Event** class, like **PI** is a property of the **Math** class, etc.), so they're accessible using code completion.

Have you noticed that constants are all in uppercase? That's to distinguish them as constants. Although capitalization is not necessary, it's considered a best practice. And because it's difficult to distinguish separate words in all-uppercase text, the standard is to place an underscore (_) between the words. Now you know.

Using `addEventListener()` in ActionScript

It's time to demonstrate some ActionScript-only event listener definitions. We're getting close to our game development at the end of this chapter, so we want these examples to pull double-duty and teach you a few tricks that we'll apply later. This way, it won't be a shock when you see them in the game code.

We'll show you two more examples before we get into the fun stuff. The first example creates a draggable green ball on the screen, and it will teach you how to create a simple FXG graphic as well as how to enable dragging on a layout component. The second example demonstrates how to set up a **Timer** to make the same ball move from one side of the screen to the other.

Creating a draggable component

Many developers like to take advantage of the **Application** container's **creationComplete** event to set up several event listeners within the same named function. In a larger application, it's more efficient to place all your event listener definitions in one area of the code. To demonstrate, Example 7-3 contains a project we created called **DragCircle** that draws a simple, draggable FXG graphic in Flash Player (Figure 7-15).

Example 7-3. *C:\Flex4Code\DragCircle\src\DragCircle.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  applicationComplete="onAppComplete()">

  <fx:Script>
    <![CDATA[
      private function onAppComplete():void{
        circle.addEventListener(MouseEvent.CLICK, dragOn);
        circle.addEventListener(MouseEvent.CLICK, dragOff);
      }

      private function dragOn(event:MouseEvent):void{
        circle.startDrag();
      }

      private function dragOff(event:MouseEvent):void{
        circle.stopDrag();
      }
    ]]>
  </fx:Script>

  <s:Group id="circle">
    <s:Ellipse height="100" width="100">
      <s:fill>
        <s:SolidColor color="#009900"/>
      </s:fill>
      <s:stroke>
        <s:SolidColorStroke color="#000000" weight="2"/>
      </s:stroke>
    </s:Ellipse>
  </s:Group>
</s:Application>
```

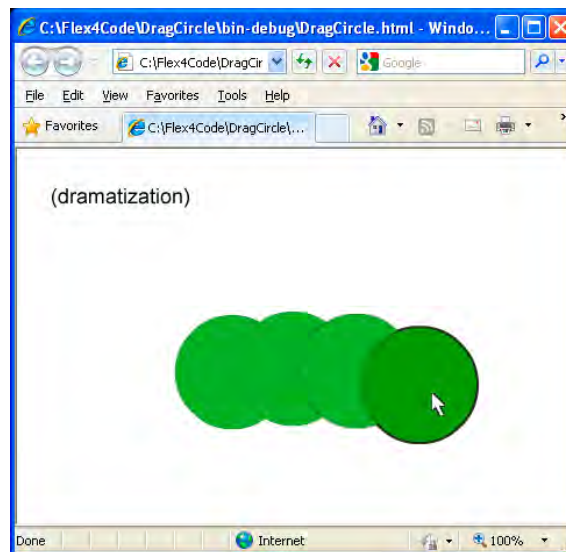


Figure 7-15. Dragging an FXG graphic

What's to notice here? Let's work from the bottom up this time. See that MXML **Group** container? The only thing it contains is our FXG graphic, which we gave an **id** of **circle**. In fact, if you flip back to Chapter 1, you'll find that this FXG graphic is nearly identical to the one we presented as an example of FXG. By now, your understanding of XML structure should be enough to make sense of the graphic.

NOTE

Throughout this example, we referred to our circle graphic as a “draggable” component; that's because we aren't managing its drop into a different container. If you have an interest in that sort of interaction, check out the following blog posting by Justin Shacklette: <http://saturnboy.com/2009/08/drag-and-drop-flex-4/>.

Moving up, we have two functions, **dragOn** and **dragOff**, practically written by code-ninja Mr. Miyagi himself. As you can see, each function toggles either **startDrag()** or **stopDrag()** on the **Group** container, which has the **id** of **circle**. This is important: the FXG graphic doesn't support **startDrag()** and **stopDrag()**, so we wrapped the graphic inside a lightweight Spark container that does support dragging.

The top of the example shows the **Application** container's **applicationComplete** event calling the named function **onAppComplete()**, which enables the event listeners, **MouseEvent.MOUSE_DOWN** and **MouseEvent.MOUSE_UP**, which start and stop the dragging, respectively.

Notice two things about the event listener definitions and the named functions. First, *the listener definitions do not include the parentheses (e.g., **dragOn()**) of their function references*. Second, both named functions pass in a **MouseEvent** class as an event variable. If you add the constructor parentheses to the called functions in the **addEventListener()** definitions, you'll get errors. Likewise, if you remove the event parameters being passed into the functions, you'll also get errors.

Creating “Invitations” with the Mouse

Here’s a solution for changing the color of the FXG graphic based upon **MouseEvent.MOUSE_OVER**. Doing so would create an “invitation” for users to interact with your draggable component. Here’s a hint, but it’s up to you to plug it into your application:

```
<fx:Script>
  <![CDATA[
    private function onAppComplete():void{
      circle.addEventListener(MouseEvent.MOUSE_OVER, onOver);
      circle.addEventListener(MouseEvent.MOUSE_OUT, onOut);
    }

    private function onOver(event:MouseEvent):void{
      fillColor.color=0x007B00;
    }

    private function onOut(event:MouseEvent):void{
      fillColor.color=0x009900;
    }
  ]]>
</fx:Script>

<s:Group id="circle">
  <s:Ellipse height="100" width="100">
    <s:fill>
      <s:SolidColor id="fillColor" color="#009900"/>
    </s:fill>
    <s:stroke>
      <s:SolidColorStroke color="#000000" weight="2"/>
    </s:stroke>
  </s:Ellipse>
</s:Group>
```

Creating a self-powered, moving component

Example 7-4, called **PoweredCircle**, teaches you the following skills:

- You can use a **Timer** class to automate events.
- The **KeyboardEvent** class allows you to catch keyboard input.
- The **Alert** class allows you to output simple messages to the user.

While you’re at it, you’ll also have a chance to rehearse the **trace()** method and review the structure of an **If** block conditional.

Foremost, we're using the same FXG graphic from the previous lesson, and we're building our event listeners in the **Application** container's **applicationComplete** handler. So we won't spend time discussing those aspects of this example. Instead, we'll focus on understanding the **Timer**, the **KeyboardEvent**, and the **Alert** classes.

Example 7-4. *C:\Flex4Code\PoweredCircle\src\PoweredCircle.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    applicationComplete="onAppComplete()">

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;

            private var timer:Timer = new Timer(10,0);

            private function onAppComplete():void{
                stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDown);
                timer.addEventListener(TimerEvent.TIMER, ballMover);
            }

            private function keyDown(event:KeyboardEvent):void{
                trace(event.toString());
                // trace shows event.keyCode as 13 for ENTER
                if(event.keyCode == 13){
                    timer.start();
                }
            }

            private function ballMover(event:TimerEvent):void{
                circle.x += 5;
                // test circle position, stop at edge
                if(circle.x + 100 >= stage.stageWidth){
                    trace("Finished");
                    timer.stop();
                    Alert.show("Finished", "PoweredCircle");
                }
            }
        ]]>
    </fx:Script>

    <s:Group id="circle" verticalCenter="0">
        <s:Ellipse height="100" width="100">
            <s:fill>
                <s:SolidColor color="#009900"/>
            </s:fill>
            <s:stroke>
                <s:SolidColorStroke color="#000000" weight="2"/>
            </s:stroke>
        </s:Ellipse>
    </s:Group>

</s:Application>
```

WARNING

Flash Builder will usually create **import** statements as you add classes in ActionScript; however, you might need to import the **Alert** class manually.

NOTE

In order to determine the proper **keyCode** value for the Enter/Return key, we set a **trace()** on the **keyDown** function's **event** parameter, converting the entire **event** object into a **String** representation. Then, we launched the application in Debug mode and hit the Enter key (we actually hit a ton of keys for the fun of it) to see which values changed.

We recommend you repeat this **trace()** step when you construct this application to reinforce your debugging skills, which are priceless when you're on your own, to research the burning question "So, what am I supposed to do with this object?"

The first thing you should notice about the **Timer** in this example is that we created it with class-level scope; the **private** keyword ensures any method in the class (the application in this case) can access properties and methods of the **Timer**. Next, the **Timer** takes two variables when it's constructed: **delay** and **repeatCount**. The **delay** property represents the number of milliseconds the **Timer** will suspend between firing its **TimerEvent**. The **repeatCount** is the number of times the **Timer** will iterate before stopping itself. We chose a **delay** of 10 milliseconds and a **repeatCount** of 0. A **repeatCount** of 0 translates to unlimited repetitions. If you look down through the code, you'll notice that the **Timer** has two methods we'll be calling, **start()** and **stop()**; these methods perform as you would expect. Further, notice that the timer's event listener calls the function **ballMover()**. We'll discuss that function in a moment, but realize that the **Timer** will call **ballMover()** once every 10 milliseconds.

Next we have the **KeyboardEvent** listener waiting for any key to be pressed. So, what's this **stage** that will listen for the **KeyboardEvent**? Here's how the Language Reference defines it: "The **stage** class represents the main drawing area." It goes on for about 10 more pages discussing the **stage**, but for our purposes, listening for keystrokes at the application level requires attaching our event listener to the **stage**. Moving on, the **KeyboardEvent** listener is calling the handler **keyDown()**. Every time a key is pressed, the **keyDown()** function tests the **event** parameter's **keyCode** property (**event.keyCode**) to see whether it equals 13, which corresponds to the Enter/Return key. If the condition evaluates to true, **keyDown** calls the timer's **start()** method, beginning the automation, or in this case, the animation of the circle graphic across the screen.

Now for the **ballMover()** function. Once the automation is engaged, **ballMover()** recalculates the circle graphic 5 pixels to the right once every 10 milliseconds. The function uses the *addition assignment operator* (**+=**) to say "your current **x** value, plus 5." After the movement is committed, the function compares the graphic's **x**-value to the application's width to detect whether the move is complete.

The **Stage** class has some useful properties, namely **stageWidth** and **stageHeight**, that tell us the dimensions of the application at runtime. Resizing the screen will create a problem—you'd need another listener to catch **Event.Resize**—but for our current needs it doesn't matter.

Back to the function: if the graphic's **x** value plus its **width** meets or exceeds the width of the **stage** (i.e., the expression **circle.x + 100 >= stage.stageWidth**, becomes true) the graphic has finished its move and the **Timer** is stopped. Next, **ballMover()** calls the **Alert** class's **show()** method to create the "Finished" message. The first parameter in the **show()** method is the **message** string; the second parameter is the **title**. If you run this application, you should end up with something pretty similar to Figure 7-16.

WARNING

When you run this application, you'll need to click on it somewhere to give it focus. Without focus, the application won't catch your keystrokes, and the animation will not begin.

NOTE

*There are a lot of useful properties and methods attached to the **Stage** class. Check it out sometime in the Language Reference documentation: <http://www.adobe.com/livedocs/flash/9.0/ActionScriptLangRefV3/flash/display/Stage.html>.*

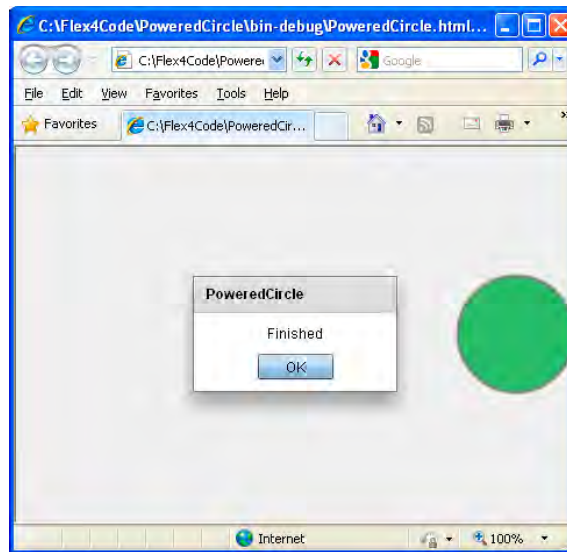


Figure 7-16. An Alert message noting the animation is complete

Collision! A Whirlwind of Events

*For Tim Bloomquist, who is always asking me to program a game,
...and for Steve Robison, who loves pinball.*

Take a deep breath, because we're about to take you right to the edge of everything you've learned about Flex. We're going to make a Flash game called *Collision!*, and in the process we'll borrow something from everything you've learned so far.

The game is based on a combination of timeless classics. It's like *Pong* in the sense that you're moving a paddle around trying to intercept a moving ball, but it's also like *pinball* in the sense that it's one-player. However, we took this hybrid concept a couple of steps further by adding some twists. For one, we added the ability to move the paddle top to bottom in addition to left and right. Second, we thought it would be cool if you could actually *drag* the paddle and play with the mouse rather than the keyboard. Surprisingly, it ends up being relatively fun to play.

Here's the good news: the project is about 190 lines of code, and this chapter prepared you by teaching almost everything you need to know coming into this exercise. We ramped you up. But, here's the bad news: you might not like the math necessary to calculate the trajectory of the ball. On the bright side, it's only four or five lines of math! That's not bad!

We'll approach this project code-first. That is, first, you'll link to the external source directory we started in Chapter 5, and then you'll create a new components package. Next, we'll have you program two supporting components, **Ball** and **Paddle**, and save them to the new package. Then, rather than bore you with a multipage onslaught of explanations, we decided to scare you first by giving you the source code for the main application file. After you've had a chance to glance over the main source, we'll spend the rest of this section explaining the details, including a breakdown of each of the five functions in this application. Five functions, five lines of math. You should be excited; this is a cool project! Let's get underway.

Create the Supporting Classes

The name of the game is *Collision!*, so start by creating a new Flex project named **Collision** in your workspace.

Linking the external source and adding a package

With the new project established, make sure it's selected in the Package Explorer, then open Project→Properties→Flex Build Path, and click the "Source path" tab.

In the Source path dialog, select Add Folder and browse to `C:\Flex4Code\com`, or the equivalent workspace directory on your system. This is the package root we created back in Chapter 5. Add the package as an external source path.

Now that the external package is linked to the project, right-click it (Windows) or Control-click it (Mac OS) and follow New→Package in the drop-down menu. When the dialog appears, it should already offer the name "learningflex4". Append the name so that we're creating the package *learningflex4.components*. Then, click Finish.

When you emerge into the editor, you should have a fresh, empty package directory, wherein we'll create the **Ball** and **Paddle** components for **Collision**.

Ball.mxml

The code for this class is nothing new, but the task of creating a new component might be awkward at first. To start the **Ball** component, right-click/Control-click your new *components* package and select New→MXML Component.

Enter the name **Ball**, and delete the values for Width and Height so those fields are empty. If the "Based on" field is not set to **spark.components.Group**, click Browse and search for that class. Your dialog should resemble Figure 7-17 when you're ready to proceed. Click Finish to commit the new component class.

NOTE

If you need a refresher on how to add an external package to your project, return to Chapter 5 and review the section titled "Packages" on page 79.

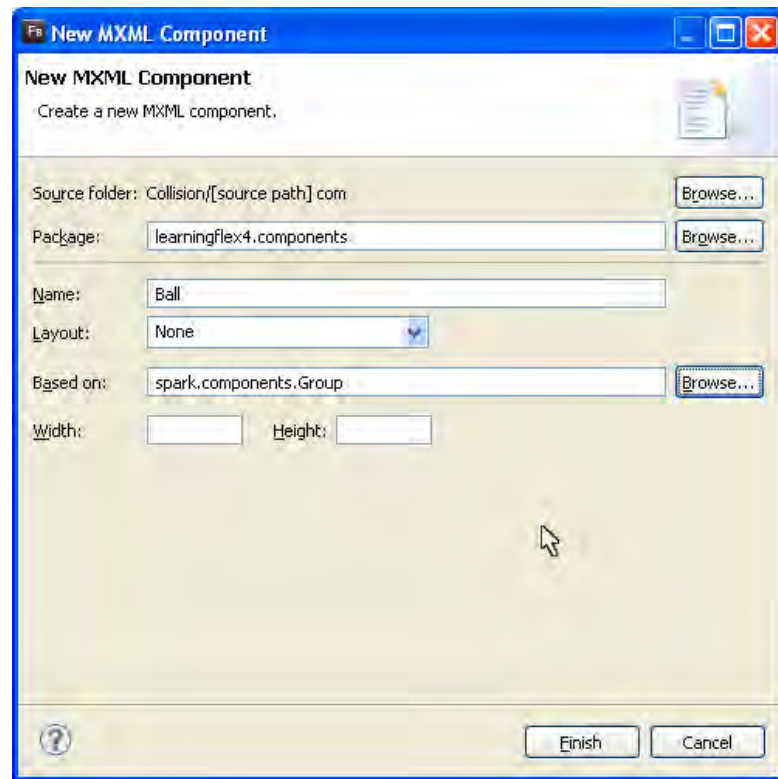


Figure 7-17. The New MXML Component dialog

You should emerge into a new editor window. Add the code in Example 7-5 and save to create the custom component.

NOTE

Notice that the root tag for your component is not an **Application** container, but a **Group** component.

Example 7-5. C:\Flex4Code\com\learningflex4\components\Ball.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Ellipse height="20" width="20">
    <s:fill>
      <s:SolidColor color="#FF0000"/>
    </s:fill>
    <s:stroke>
      <s:SolidColorStroke color="#000000" weight="1"/>
    </s:stroke>
  </s:Ellipse>

</s:Group>
```

Paddle.mxml

Now we need to create the **Paddle** component. Repeat the tasks you performed in the previous step to create another **Group**-based component called **Paddle**.

When you emerge into the editor, add the code in Example 7-6 to create the **Paddle** component. This code is similar to Example 7-3, **DragCircle**, which we created earlier in this chapter. Notice that the **Paddle** component establishes its event listeners in the **initialize** event handler. The component's **creationComplete** handler would work equally well, but we used the **initialize** event in this case. When you're finished entering the code, save the component to commit it to the project. If you don't save, code completion might not see your new component when we're drafting the main application source.

Example 7-6. *C:\Flex4Code\com\learningflex4\components\Paddle.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  initialize="onInit()">

  <fx:Script>
    <![CDATA[

      private function onInit():void{
        addEventListener(MouseEvent.CLICK, dragOn);
        addEventListener(MouseEvent.CLICK, dragOff);
      }

      private function dragOn(event:MouseEvent):void{
        startDrag();
      }

      private function dragOff(event:MouseEvent):void{
        stopDrag();
      }

    ]]>
  </fx:Script>

  <s:Rect height="10" width="100">
    <s:fill>
      <s:SolidColor color="#0000FF"/>
    </s:fill>
    <s:stroke>
      <s:SolidColorStroke color="#000000" weight="1"/>
    </s:stroke>
  </s:Rect>

</s:Group>
```

Once you have the **Paddle** component built and saved, let's test it to make sure it will support dragging like we expect. The quickest solution is to add the component to the main application, compile it, and test it. So jump back to the **Collision** editor.

Notice Figure 7-18. If you simply type `<padd`, code completion should kick in and offer you the class. Either double-click the response or press Tab and then Enter/Return to add the component to your project. If your **Collision** code looks something like Example 7-7, go ahead and run it and try dragging the **Paddle** around the screen.

Example 7-7. Ten lines of code and already testing

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:components="learningflex4.components.*">

  <components:Paddle/>

</s:Application>
```

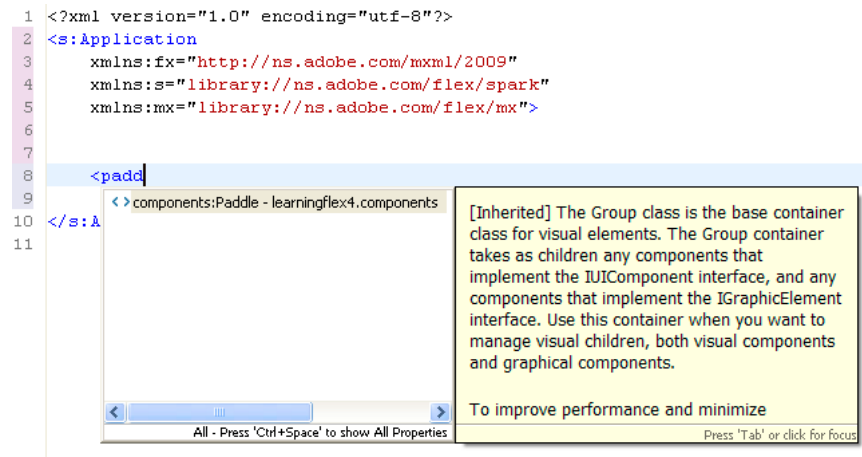


Figure 7-18. Adding the *Paddle* using code completion

The Main Application Code

As promised, here's the main application code for **Collision**, shown in Example 7-8. Depending on your preferences, either study this until you feel content or simply jump down to the next section and peruse our explanations. Of course, if you want to start entering this code into the main application immediately, go for it!

In the best-case scenario, you're looking at this code and feeling pretty good about it, and maybe you intend to read only a few of our explanations. On the other hand, maybe you're dreading it and thinking, "Ack." Either way, take an approach that makes you comfortable. Just don't give up! You're closer than you think to a major sense of accomplishment.

Example 7-8. *C:\Flex4Code\Collision\src\Collision.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:components="learningflex4.components.*"
  applicationComplete="onAppComplete()"
  height="400" width="500">

  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;

      private var timer:Timer = new Timer(10,0);
      private var xStart:Number = 0;
      private var yStart:Number = 0;
      private var radius:uint = 0;
      private var degrees:uint = 45;
      private var hitCount:uint = 0;
      private var speedFactor:uint = 5;
      private var levelCount:uint = 1;

      private function onAppComplete():void{
        // listen for keyboard interaction
        stage.addEventListener(KeyboardEvent.KEY_DOWN, keyDown);
        timer.addEventListener(TimerEvent.TIMER, ballMover);

        resetGame();
      }

      private function resetGame():void{
        // reset timer and ball position..
        timer.stop();
        ball.x = 0;
        ball.y = 0;
        // reset paddle position..
        paddle.x = Number(stage.stageWidth / 2) - 50;
        paddle.y = Number(stage.stageHeight - 50);
        // reset movement variables..
        xStart = 0;
        yStart = 0;
        radius = 0;
        degrees = 45;
        // reset game play variables..
        hitCount = 0;
        speedFactor = 5;
        levelCount = 1;
        levelLabel.text = "Level: 1";
      }
    ]]>
  </fx:Script>
</s:Application>
```

```

private function startBallMovement():void{
    timer.start();
}

private function ballMover(event:TimerEvent):void{
    // move ball first..
    //http://www.actionscript.org/forums/showthread.php3?t=177361
    var radians:Number = degrees * Math.PI/180;
    radius += speedFactor;
    ball.x = radius * Math.cos(radians) + xStart;
    ball.y = radius * Math.sin(radians) + yStart;

    // test for collision with paddle, 2 parts..
    // test for collisions with right, top, left..
    // if collision, consider level increase..
    // if no collision, test for failure..
    if (((ball.x + 20 >= paddle.x && ball.x + 20 <= paddle.x + 100)
        &&
        (ball.y + 20 >= paddle.y))
        ||
        ball.x + 20 >= stage.stageWidth
        ||
        ball.y + 20 <= 0
        ||
        ball.x + 20 <= 0)
    {
        trace("collision!");
        degrees += 90;
        xStart = ball.x;
        yStart = ball.y;
        radius = 0;
        hitCount += 1;
        if(hitCount == 10){
            // level increase..
            hitCount = 0;
            speedFactor ++;
            levelCount ++;
            levellabel.text = "Level: " + levelCount;
        }
    }else{
        // test for failure..
        if(ball.y + 20 >= stage.stageHeight){
            trace("failed!");
            timer.stop();
            Alert.show("YOU HAVE LOSS!", "Collision!");
            paddle.stopDrag();
        }
    }
}

private function keyDown(event:KeyboardEvent):void{
    switch(event.keyCode){
        case 37: //key left
            paddle.x -= 20;
            break;
        case 39: //key right
            paddle.x += 20;
            break;
        case 38: //key up
            paddle.y -= 20;

```

```

        break;
    case 40: //key down
        paddle.y += 20;
        break;
    }
}

]]>
</fx:Script>

<s:Rect height="100%" width="100%">
    <s:stroke>
        <s:SolidColorStroke color="#000000" weight="1"/>
    </s:stroke>
</s:Rect>

<s:Label id="titleLabel" top="5" horizontalCenter="0"
    text="Collision!" fontWeight="bold"/>

<s:Label id="levelLabel" top="5" right="5"
    text="Level: 1" fontWeight="bold"/>

<s:Button label="start" bottom="2" left="2"
    click="startBallMovement()"/>

<s:Button label="reset" bottom="2" right="2"
    click="resetGame()"/>

<components:Paddle id="paddle"/>

<components:Ball id="ball"/>

</s:Application>

```

Initial Insights

Some observations may leap out at you immediately. For instance, there's a new namespace in this code. The math looks fairly intense, and that's one cryptic **If** block. Switch? Case? What is **Switch..Case**? We break down these walls for you in this section.

A new namespace

Did you notice the deviation emphasized in Example 7-9 among the namespace declarations?

Example 7-9. *The new namespace*

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:components="learningflex4.components.*"
    applicationComplete="onAppComplete()"
    height="400" width="500">

```


If you need a primer, we discussed namespaces in Chapter 4 in the section “S, FX, and MX: Namespaces Explained” on page 60. However, it’s not a mystery. The **components** namespace references the components in our custom *components* package, and it corresponds directly to the package name. If you don’t like the namespace **components**, you’re at liberty to change it to any unique namespace you prefer.

The math

How about the first few lines in the **ballMover()** function, shown in Example 7-10?

Example 7-10. *The math passage, which determines the trajectory of the Ball*

```
private function ballMover(event:TimerEvent):void{
    // move ball first..
    //http://www.actionscript.org/forums/showthread.php3?t=177361
    var radians:Number = degrees * Math.PI/180;
    radius += speedFactor;
    ball.x = radius * Math.cos(radians) + xStart;
    ball.y = radius * Math.sin(radians) + yStart;
```

NOTE

If the URL is difficult to read among the source code, here’s the link to the forum where we found the math-magic solution to calculate the **Ball** component’s trajectory: <http://www.actionscript.org/forums/showthread.php3?t=177361>.

Converting between radians and degrees? **Math.cos**? **Math.sin**? Whoa! Don’t worry; we won’t turn this into a trigonometry lesson. In fact, we can’t claim this math; it’s a gem gleaned from the user community and adapted for this application. Notice that we included the forum URL as a comment right in the source code, as if to say, “Go here for some extra insight into this code.” If you follow that URL, however, you’ll realize that we did impose some hacking to get the results we wanted, particularly with regard to the data types used and the **speedFactor** variable, which we added as a means of increasing the game speed as the **Ball**’s collision count increased.

The branching structures (decision logic)

This application has two significant decision-making, or *branching*, routines.

The first decision considers a complex set of **Boolean** criteria (it’s either true or false) to determine whether a collision has occurred. The decision is evaluated with every movement of the **Ball** (every 10 milliseconds).

The second decision routine, a **Switch..Case** block, toggles **Paddle** movement using the keyboard arrow keys. We haven’t discussed a **Switch..Case** routine, but if we can help you comprehend the collision-detection conditional, which we’ll explain in the next section, you won’t have any trouble with the **Switch..Case** block. They’re actually quite clean and a joy to work with, as opposed to many nested **If..Else** blocks.

The Five Functions

This section considers the five functions necessary for **Collision** to create game play.

onAppComplete()

Like many of our prior examples, **onAppComplete()** is used to define event listeners for both **KeyboardEvent.KEY_DOWN** and **TimerEvent.TIMER**. The former forces the application to listen for keyboard interaction, and the latter is used to animate the **Ball** component by updating its position every 10 milliseconds.

This function also calls **resetGame()** on startup, mostly just to set the **Paddle**'s starting position.

resetGame()

This function is called by both **onAppComplete()** and the reset **Button**.

The purpose of this function is to reset the **Timer**, the **Ball** position, the **Paddle** position, the movement variables, and the game play variables back to their initial values, which are assigned near the top of the application code.

The technique used to reset the **Paddle** is compelling, particularly the **Paddle**'s **x** value, as it takes half of the **stageWidth** value and then subtracts half of the **Paddle**'s **width** (50) in order to arrive at a true center placement:

```
paddle.x = Number(stage.stageWidth / 2) - 50;  
paddle.y = Number(stage.stageHeight - 50);
```

startBallMovement()

This function is called by the start button **click** event. In turn, it calls the **Timer** class's **start()** method, which begins the **Ball** animation, which effectively begins the game.

ballMover()

The **ballMover()** function is the engine that drives the entire game. It calculates the trajectory of the **Ball**, tests the **Ball**'s position for collisions, handles collision responses, and, when necessary, ends the game—and it will be necessary.

Calculating the trajectory

Here's how we explain the **Ball**'s trajectory calculation. (Sorry, we lied when we said we weren't going to give you a math lesson.)

For the **Ball** to move along a path, we need to establish its origin, speed, and direction. We can choose any starting position and any start angle. Furthermore, the **Timer** can establish speed as a function of pixels per 10

milliseconds. So here is a restatement of the givens according to the values we've selected:

Point of origin

`xStart/yStart = 0,0`

Initial direction of travel

45 degrees

Rate of travel

Initially, 5 pixels / 10 milliseconds

To reinterpret Xegnama's advice from the ActionScript.org forum, we can use trigonometric functions to calculate points along the perimeter of a circle that increases in size for every subsequent measurement. We can grow the circle by increasing its **radius**, thus allowing for calculation of a new point beyond the last.

If you search for documentation on this subject, you'll find the following equations, appearing together as an approach for calculating a point on a circle as `x/y`, where we know the origin of the circle (`xStart/yStart`) and `t`, an angle in radians:

`newX = xStart + (radius * cosine(t))`

`newY = yStart + (radius * sine(t))`

Documentation also tells us that we can convert from degrees to radians:

`t = degrees * pi/180;`

Backed by these ideas, we can achieve increasing speed of trajectory by increasing the circle size, not at a fixed rate, but through a factored increase of its radius. We found that a modifier of +1 to the radius (**speedFactor**) once every 10 collisions created a natural rate of acceleration.

With that said, here's the math routine that calculates the trajectory:

```
var radians:Number = degrees * Math.PI/180;
radius += speedFactor;
ball.x = radius * Math.cos(radians) + xStart;
ball.y = radius * Math.sin(radians) + yStart;
```

With each level-up, this line of code modifies the circle to produce acceleration; notice the use of the *incrementing operator* (`++`) to increase the **speedFactor** by 1, modifying the radius as a number of pixels:

```
speedFactor ++;
```

Testing for collisions

The first decision in the collision-detection routine is a fairly dense **Boolean** condition (i.e., it's either true or false) that tests the **Ball**'s position for collisions with the **Paddle** or the **stage** boundaries. We're throwing some new operators at you here, including the greater-than or equal (`>=`) operator, the

WARNING

These lines represent trigonometric formulas, not ActionScript code.

NOTE

*The Flex framework provides the trigonometric functions as **Math.cos** (cosine) and **Math.sin** (sine), and **Math.PI** as the constant pi.*

AND operator (**&&**), the less-than or equal (**<=**) operator, and the **OR** operator (**||**).

While you consider this routine, remember *any* of these four criteria testing positive means a collision has occurred.

The first criteria tests for a collision between the **Ball** and the **Paddle**. It looks at two expressions. The first expression tests whether the **Ball**'s **x**-value (upper-left corner plus a width of 20) falls within the **x**-range occupied by the **Paddle**, which extends between the **Paddle**'s **x**-value and its width of 100—hence greater or equal (**>=**). If true, there is an intersection of the **x**-axis. However, we also (**&&**) need an intersection of the **y**-axis. The second expression compares the **Ball**'s **y**-value plus its depth of 20 to the **Paddle**'s pure **y**-value, representing the plane of the **Paddle**'s surface. If *both* of these criteria are true, there is a collision with the **Paddle**. There's no shame in reading that paragraph a couple of times to try and grasp this.

The most difficult aspect of this criteria to keep straight is the pattern of parentheses used to distinguish the two halves of the condition, as well as the parentheses used to unite the two expressions into a single criteria. In the code in Example 7-11, the first expression represents the **x**-axis check, and the second expression represents the **y**-axis check. If we divorce the first line from the opening conditional parenthesis and emphasize the individual expressions, it may be easier to see the structure of the two-part conditional.

Example 7-11. *Testing for collisions with the Paddle*

```
If (
    ((ball.x + 20 >= paddle.x && ball.x + 20 <= paddle.x + 100)
    &&
    (ball.y + 20 >= paddle.y))
```

The next three criteria, differentiated from one another by the **OR** operator (**||**) and shown in Example 7-12, test for collisions with the right, top, and left borders. You'll find these three criteria are similar to the one used in Example 7-4, that stopped the **PoweredCircle** when it hit the **stage** boundary. To test the right boundary (line three of the conditional), the routine compares the **Ball**'s **x**-value plus its width to the **stageWidth**; if the **Ball**'s rightmost **x**-value has met or exceeded **stageWidth**, there is a collision. The top border (line four) is much easier. If the **Ball**'s unadjusted **y**-value meets or goes below the origin (0), there is a collision. And finally (line five), if the **Ball**'s **x**-value plus its width is less than or equal to the origin (0), there is an intersection.

Example 7-12. *Testing for collisions with the borders*

```
//continuing..
||
ball.x + 20 >= stage.stageWidth
||
ball.y + 20 <= 0
||
ball.x + 20 <= 0)
{
```

Handling collisions

If a collision is detected, much like analyzing the conditions of a billiards table, we modify the response trajectory of the **Ball** by adding 90 to **degrees**. We also move the imaginary circle, which is used to calculate trajectory, to a new position (**xStart/yStart**) equaling the **Ball**'s location at the moment of the collision. Next, we reset the **radius** to zero and increment the **hitCount**, which processes a leveling-up every 10 collisions.

```
trace("collision!");
degrees += 90;
xStart = ball.x;
yStart = ball.y;
radius = 0;
hitCount += 1;
```

Leveling-up gradually increases the speed of the game by incrementing (**++**), or adding 1, to a **speedFactor** variable after every 10 collisions. The **speedFactor** subtly affects the trajectory calculation so that the **Ball** moves faster through the course of the game. The game also tracks the current level rating and outputs it to a **Label** control:

```
if(hitCount == 10){
    // level increase..
    hitCount = 0;
    speedFactor ++;
    levelCount ++;
    levelLabel.text = "Level: " + levelCount;
}
```

Handling game over

If no collision is detected, a final check is made to see whether the **Ball** collides with the bottom of the **stage**, signaling game over. The test is simple: if the **Ball**'s **y**-value plus its depth meets or exceeds **stageHeight**, the failure criteria are achieved.

Game over results in stopping the **Timer** and alerting the user of their defeat, for which we couldn't resist using a message reminiscent of the 8-bit days, when Nintendo games like *Metal Gear* ("Uh oh! The truck have started to move!") and unbeatable-without-cheating *Contra* were in their glory days.

```
if(ball.y + 20 >= stage.stageHeight){
    trace("failed!");
    timer.stop();
    Alert.show("YOU HAVE LOSS!", "Collision!");
    paddle.stopDrag();
}
```

keyDown()

The **keyDown()** function handles the **KeyboardEvent.KEY_DOWN** event response:

```
private function keyDown(event:KeyboardEvent):void{
    switch(event.keyCode){
        case 37: //key left
            paddle.x -= 20;
```

```

        break;
    case 39: //key right
        paddle.x += 20;
        break;
    case 38: //key up
        paddle.y -= 20;
        break;
    case 40: //key down
        paddle.y += 20;
        break;
    default:
        //do nothing
        break;
    }
}

```

The **Switch..Case** block is particularly elegant because it remains readable in spite of having many conditions. Moreover, this sort of conditional block is easily expanded when new conditions need to be added. An example would be adding support for additional keyboard keys, such as the Enter/Return key, which might be used as an alternate method to start the game. For more information about **Switch..Case** statements, see the sidebar titled “Making Decisions with Switch..Case” on page 134.

The **keyCode** value for each arrow key was identified using the **trace()** technique recommended in Example 7-4. Depending upon which arrow key is depressed, the **Paddle** moves either 20 pixels left, right, up, or down, as indicated by comments. This block makes use of the *subtraction assignment operator* (**-=**) as well as the *addition assignment operator* (**+=**), which, relatively speaking, either subtracts from or adds to the **Paddle**’s positional value at the time of the decision.

The **break** keyword tells the runtime that if a condition evaluates to true, quit processing, as there will not be another match. In other words, a keystroke won’t be interpreted as both a left and then a right arrow key; it’s one or the other.

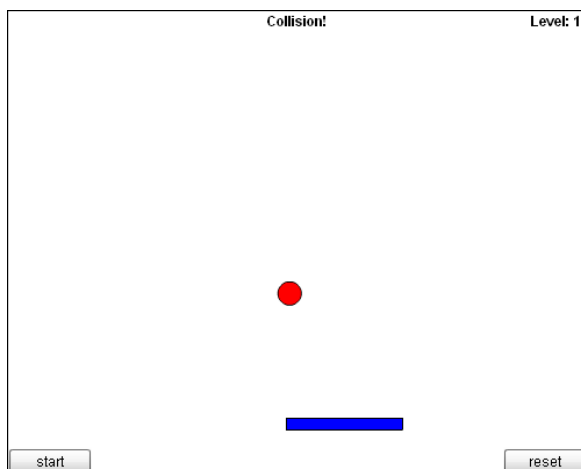


Figure 7-19. Playing Collision!

Making Decisions with Switch..Case

If you have prior programming experience, you might recognize the **Switch..Case** branching routine. *Branching* refers to the flow of information through a logical routine for the purpose of evaluating criteria and arriving at a decision.

There are some advantages to using the **Switch..Case** block over several nested **If..Else** blocks. To start, **Switch..Case** is more efficient than a series of **If..Else** statements because, thanks to the **break** keyword, you can abruptly leave the routine as soon as a condition evaluates to true. Also, **Switch..Case** is easier to read, especially when several conditions are handled together in the same block. Another advantage provided by **Switch..Case** is the ability to easily and cleanly add new conditions to a routine undergoing maintenance.

A **Switch..Case** block begins with the **switch** keyword, which is followed by a *condition* wrapped in parenthesis. Next, a series of **case** statements—notice the colon (:) following each case—provides arguments to evaluate the condition. When a match is found, the **break** keyword instructs the program to abort the remaining cases. However, if **break** is omitted, the routine continues “falling through” the remaining conditions looking for additional matches. This can be useful for processes that have many criteria. Finally, a **default** condition should always terminate a **Switch..Case** block.

The following example considers a **Switch..Case** block that evaluates city names according to the American state that contains them.

```
switch(cityName){
  case "Washington D. C.":
    return "District of Columbia";
    break;

  case "Los Angeles":
  case "San Jose":
  case "San Diego":
  case "San Francisco":
  case "Oakland":
  case "Long Beach":
    return "California";
    break;

  default:
    return "";
}
```

As we can see from this example, “District of Columbia” has only one match. So we want the program to terminate the routine after hitting that match.

On the other hand, California has many cities, so we can take advantage of **Switch..Case** handling and arrange the cases so that like criteria share the same result.

Since we don’t have a reasonable default condition, we merely return an empty string (“”).

Game Play and Customizing Strategies

This example was fun to put together, and we hope you got the code working and have enjoyed playing the game. Of course, every program deserves to be personalized by its creator, so we came up with a few suggestions to inspire you.

Basic customizing variables

The fastest hacks that could most significantly affect game play reside in how you handle the initial positioning and trajectory of the **Ball**. For example, rather than starting each session from 0,0 with a 45-degree trajectory, could you randomize the start position of the **Ball** so that it’s always at **y=0** but its **x**-value fluctuates? Similarly, could you use a randomized trajectory—say, between 35 and 60 degrees? Such modifications would be a great exercise for you, especially on the heels of using the **Math** class. If you were to add both these suggestions, game play on your version would be significantly more diverse, and in a game like this, more diverse means more fun.

Opportunities for improved performance

Linus' Law: "Given enough eyeballs, all bugs are shallow."

—Eric S. Raymond,
The Cathedral and the Bazaar

Raymond's observation could be restated like so: a bug overlooked by one programmer may be identified and fixed quickly by another, particularly one with a fresh take on the problem. As we create programs, we risk becoming too familiar with our own interpretation of the problem and its solution, and the creativity that inspired the project makes you guilty of filling in small, seemingly benign details as a by-product of having the great vision.

That philosophical musing aside, *Collision!* definitely has room for improvement, particularly with regard to collision detection and UI design. Here's our wish list:

Better collision detection

After playing a few rounds, you'll notice some bugs in the collision-detection routine. You have the unique advantage of fresh eyes, so perhaps you'll catch something and improve it. We have our suspicions, but we're going to cut it loose and see what you come up with. Don't forget to swing by <http://learningflex4.wordpress.com> to share your insights.

Create an "invitation" to drag the Paddle

The **Paddle** could use an "invitation," signaling to the user that it's accessible by the mouse. For some insight on how to accomplish this, see the sidebar titled "Creating 'Invitations' with the Mouse" on page 117.

Automatic reset after a loss

It might be better if the game automatically reset following a loss. It becomes tedious to manually reset the game after each short session. However, we also felt like there was a sort of "taunt" implied by leaving the conditions of the failure onscreen when the "game over" message appears. Can you think of a solution that might accommodate the best of both worlds?

Add a scorekeeping system

This could be a fun exercise, especially after you finish Chapter 8, which discusses data binding. Data binding would be quite suitable for devising a scorekeeping system. However, if you wanted to save the top scores, you should look into the Flash version of the "cookie," called a *shared object*. If you're interested, this post at the "Republic of Code" may answer all your questions: <http://www.republicofcode.com/tutorials/flash/as3sharedobject/>.

Add additional component classes

You could run forever with this idea. We're thinking of those obstacles that exist inside pinball machines, as well as a variant of this game that has blocks you try to break in order to move up a level. Fair warning: if you try creating a destructible block class, you'll need to retool the collision-detection system.

Automate game play when not in use

Haha, OK, we're done here.

Summary

Some chapter, eh? Zooming out, in this chapter we learned the following:

- Many common events are available to Flex UI components.
- You can use the Properties pane, code completion, or Adobe's Language Reference to research the events available to any class or component.
- You can call event handler functions using inline script, but you can use the `addEventListener()` method to define event handlers in ActionScript.
- When defining event listeners in ActionScript, reference events using their event constants (e.g., `MouseEvent.CLICK`).
- The `Alert` class can be useful for communicating simple messages to the user.

Along the way you also received introductions to the Flash coordinate system, the `Math` class, and logical decision-making using `If..Else` and `Switch..Case` blocks. You even reviewed package development concepts and created some reusable custom components. Oh, we also developed a simple yet fully playable game.

What a chapter!

In Chapter 8, we finally discuss this data binding we keep telling you about.

USING DATA BINDING

“Some things never change.”

—Popular expression

It’s difficult to imagine a future where data could be more dynamic than it is in the early 21st century. Here’s a compelling thought: what percentage of the information in our lives can we classify according to its *timestamp*?

Like data in the world, data in our applications also changes quickly. Sometimes those changes originate on the Internet in the form of stock quotes, weather information, RSS feeds, even tweets. Yet, at other times, data dynamics come entirely from our own creations. Whatever the case, one thing is certain: you want your applications to listen for and react to such changes in data.

Fortunately, thanks to something called *data binding*, it’s easy to work with dynamic data in Flex. As you’ll soon see, data binding is easy to implement, and once you’re comfortable with it, you’ll find yourself using it frequently as a means of handling data in your applications.

What Is Data Binding?

By its name, *data binding* implies multiple values bound together so that a change to one value affects the other. Data binding typically occurs in one direction and involves unique **source** and **destination** values. If the **source** value changes, data binding imposes changes on the **destination** value. Flex also offers two-way, or bidirectional, data binding. For two-way data bindings, either side of a binding statement can behave as the source or the destination.

You can implement data binding using either inline ActionScript or the special `<fx:Binding/>` tag, and in this chapter we cover examples of both.

IN THIS CHAPTER

- What is Data Binding?
- Applying Data Binding
 - Two-Way Bindings
- Handling Complex Data with Data Models
- When Data Binding Isn’t Appropriate
- Summary

Applying Data Binding

The following section demonstrates common implementations of data binding. Because the syntax allows a few different options, you can choose your preferred approach when programming data binding into your own applications.

One-Way Binding

In the most basic usage of data binding, a class property or a variable is transferred to another class property or variable. Example 8-1 demonstrates inline data binding, and the `text` property of `nameTI` is bound to the `text` property of another `TextInput` control.

Enter your full name..

Figure 8-1. Simple one-way, inline data binding

WARNING

Don't bind the **destination** property to the **source** control itself; rather, bind to the **source** control's `text` property. The following is incorrect, and although this code would not throw an error, the result would be less than spectacular:

```
<s:TextInput id="nameTI"/>
<s:TextInput editable="false"
    text="{ nameTI }"/>
```

Example 8-1. One-way, inline data binding

```
<s:VGroup horizontalAlign="center" horizontalCenter="0" verticalCenter="0">
  <s:Label text="Enter your full name.."/>
  <s:TextInput id="nameTI"/>
  <s:TextInput editable="false" text="{ nameTI.text }"/>
</s:VGroup>
```

Let's consider the syntax. First, the component representing the **source** must have a valid `id` attribute. Second, the binding recipient—the **destination**—is defined by curly braces (`{` and `}`), which wrap the **source**. Whitespace inside the braces is ignored when the property is rendered.

Within the MXML attribute definition, curly braces create the data binding. Without them, the destination component's `text` property would become `nameTI.text`. This is the simplest and most common usage of data binding in Flex.

As you can see in Figure 8-1, the text for both components is identical. Changes to the `text` property of `nameTI` transferred automatically to the second `TextInput`.

Binding Variables

Example 8-2 demonstrates one-way binding between a **String** variable in a Script/CDATA block to a `TextInput` control's `text` property.

Example 8-2. One-way binding between a variable and a control property

```
<fx:Script>
  <![CDATA[
    [Bindable]
    private var fullName:String = "Alaric Cole";
  ]]>
</fx:Script>

<s:VGroup horizontalAlign="center" horizontalCenter="0" verticalCenter="0">
  <s:Label text="Behold your full name.."/>
  <s:TextInput editable="false" text="{ fullName }"/>
</s:VGroup>
```

Multiple Applications in a Project

Flash Builder will support multiple applications in a Flex project. To create another application within an existing project, select a project in the Package Explorer pane and open File→New→MXML Application. Just complete the dialog to create a new application.

However, while a Flex project can have multiple application files, one application must be the **default application**. The default application is the application launched by the green Run button when an application is not selected/highlighted in the Package Explorer. When a project has more than one application, the Run button does consider a few criteria before deciding which application to launch:

- If an application is being edited in Design or Source mode, clicking the Run button launches that application.
- If nothing is being edited, the Run button launches the application selected in the Package Explorer pane.
- If nothing is selected in the Package Explorer, the Run button launches the default application.

Note also the drop-down list next to the Run button. This list includes every application contained in the selected/active Flex project. So, if you have more than one application in a project, you can use the Run button's drop-down menu to select the application you want to launch.

The **[Bindable]** *meta tag*, or metadata tag, identifies the **String** variable **fullName** as a **source** for data binding. Again, braces in the MXML component's attribute declaration tell the Flex compiler to bind the control's **text** property to the variable.

Without the braces, Flex would assign the literal string **fullName** as the text of the control. Like with the first example, the braces tell Flex to listen for changes to the variable's value. So if **Alaric Cole** were changed to **Elijah Robison**, the destination value would update itself accordingly.

Conversely, if the **[Bindable]** meta tag did not precede the variable definition in the Script/CDATA block, the variable would be recognized for its initial value, but changes to the variable would not be reflected elsewhere in the application.

So, why is the bindable meta tag necessary? When you create bindings, either through a tag or via script, Flex writes a lot of ActionScript behind the scenes, creating listeners for changes to the variable's value. Without explicit instructions relating what is and what isn't bindable, Flex would create a bunch of useless code for variables that you may not want to be bindable, and that code would make your application bloated and could potentially hurt performance.

Multiple Destinations

Nothing limits binding expressions to a single destination. Example 8-3 demonstrates a binding between one source, the `text` property of `mainTI`, and three destinations, `TI1`, `TI2`, and `TI3`. See Figure 8-2 for a demonstration of how it looks while running.

Example 8-3. Binding a single source value to three destinations

```
<s:VGroup verticalCenter="0" horizontalCenter="0" horizontalAlign="right">

    <s:Label text="One, Two, Three times a lady:" fontWeight="bold"/>

    <s:TextInput id="mainTI"/>

    <mx:FormItem label="1:" paddingTop="25">
        <s:TextInput id="TI1" text="{mainTI.text}"/>
    </mx:FormItem>

    <mx:FormItem label="2:">
        <s:TextInput id="TI2" text="{mainTI.text}"/>
    </mx:FormItem>

    <mx:FormItem label="3:">
        <s:TextInput id="TI3" text="{mainTI.text}"/>
    </mx:FormItem>

</s:VGroup>
```

One, Two, Three times a lady:

- 1:
- 2:
- 3:

Figure 8-2. Binding a source to multiple destinations

Concatenation

While we're on the topics of data binding and dynamic text, it's pertinent to discuss *concatenation*. We discussed concatenation previously in Chapter 5, but it makes sense to review the topic within the context of data binding. As you'll recall, concatenation is combining multiple strings of text in a piece-meal fashion.

Obviously, data binding helps manage dynamic, changing values throughout an application. So, if we designed a site that required a user login, we might want to include the user's name in select locations throughout the application.

Example 8-4 demonstrates bindings in a concatenated greeting, and Figure 8-3 shows how it looks.

Example 8-4. Bindings and inline concatenation

```
<fx:Script>
    <![CDATA[

        // These first and last names would be
        // provided dynamically by the server.
        [Bindable]
        private var firstName:String = "Jack";
```

Jack Wallace's Projects and Tasks: Hello Jack!

Figure 8-3. Concatenated text after rendering

```

[Bindable]
private var lastName:String = "Wallace";

]]>
</fx:Script>

<s:HGroup horizontalCenter="0" verticalCenter="0" fontWeight="bold">
  <s:Label text="{firstName} {lastName}'s Projects and Tasks:"/>
  <s:Label text="Hello {firstName}!"/>
</s:HGroup>

```

As noted by the script comments, we assume the `firstName` and `lastName` variables will be assigned dynamically by the server after a user login.

NOTE

We're also showing you a layout trick in Example 8-4. Notice how applying `fontWeight="bold"` to the `HGroup` passed that style down to its children. We discuss layouts in the next chapter.

Whitespace and inline bindings

In Example 8-4, the first concatenation inserts `firstName` and `lastName` to create a greeting. Note how the compiler preserved whitespace inside the quotation marks. The second concatenation also uses whitespace to render the human-friendly greeting. Figure 8-3 illustrated the result.

However, the following modifications to our code, shown in Example 8-5, would produce *identical* results. Note the extra whitespace inside the binding braces.

Example 8-5. Demonstrating whitespace and inline bindings

```

<s:Label text="{ { firstName } { lastName }'s Projects and Tasks:"/>

<s:Label text="Hello { firstName }!"/>

```

The Flex compiler ignores whitespace within inline expressions, so these changes won't compromise the result. Stay with us here, because we're building up to the “take-home message.”

```

<s:Label text="Hello, { firstName } { lastName }"/>

```

NOTE

Some Flex developers like to add spaces within inline expressions to improve readability. Whether it helps or not is your call.

ActionScript between the braces

There's another approach to consider regarding inline concatenation. The previous example took advantage of whitespace and literal text within the double quotes of an MXML property assignment. As Example 8-6 demonstrates, however, the entire concatenation can be handled using bindings and inline ActionScript. Single quotes (') distinguish literal text from the variable bindings.

Example 8-6. A scripted, inline concatenation using bindings

```

<s:Label text="{firstName + ' ' + lastName + '\s Projects and Tasks:'}"/>

<s:Label text="{ 'Hello ' + firstName + '!' }"/>

```


A handful of differences exist between this and our former approach. Whitespace between the braces is still ignored, but now we have to explicitly create spacing within the concatenation, as we did using the *addition operator* (+) followed by a spacing string (' ').

Does this method of concatenating a text string inside braces seem familiar? It should, because it's *ActionScript*! When you build a concatenation like the one in Example 8-6, you're actually using more inline *ActionScript*. Everything happening between the braces is *ActionScript*.

Garbage In, Garbage Out

Here's a goofy demonstration of *ActionScript* between the braces:

```
<s:Label text="Hello {firstName}!" />
```

It's *ActionScript* comment syntax, where the compiler expects an inline expression. It'll throw an error if you try to compile it, so it's fully useless, but it's funny. The Flex parser couldn't be more confused! It knows script is allowed between the braces, but it can't close the line, presumably because the comment forces it to ignore the right brace.

Escape Sequences

In the previous example, did you notice the backslash (\) character within the concatenation statement? The combination of a backslash followed by the single quote (\'), shown in Example 8-7, is referred to as an *escape sequence*.

Example 8-7. *Emphasizing the escape sequence*

```
<s:Label text="{firstName + ' ' + lastName + '\s Projects and Tasks:'}" />
```

Consider the potential problem at hand. Our concatenated phrase requires the apostrophe to establish the grammatical possessive (i.e., “Jack’s”). Meanwhile, the Flex parser is also looking for apostrophes to distinguish literal text from binding variables. This is where the escape sequence becomes necessary.

The backslash followed by the single quote does something similar to a Jedi mind trick, effectively telling the compiler “this is not the single quote you’re looking for” and allowing the apostrophe to “go about its business” and become included in the concatenation.

Any time single or double quotes are used to enclose text, the quote symbol that forms the enclosure cannot be used within the enclosure without being preceded by a backslash, forming the escape sequence. Both \' and \" are valid escape sequences as long as the enclosing symbols are single or double quotes, respectively.

You Can Quote Me on This

Best practices recommend using single quotes between braces in a binding expression. This syntax prevents confusion with the double quotes used by the XML attribute. When the Flex compiler sees a pair of double quotes, it assumes the attribute is complete, and if extra double quotes are hanging around, the compiler gets confused.

However, in some situations it might be beneficial to reverse the double and single quotes, like this:

```
text="{\"Hello, \" + firstName + \" \" + lastName}"
```

You can also use Flash Builder's Design mode when you need help creating the proper syntax. Select a component in Design mode, and then modify its **text** property in the Properties pane. When you enter a value or binding expression in Design mode, Flash Builder will ensure proper characters are used, possibly inserting escape characters where necessary. When you enter quotes in Design mode, Flash Builder might use a character entity such as `"`, which means "double quote." Special character sequences like these act as a substitute for other characters and prevent conflicts in your code.

Resizable Application Layouts

Data bindings don't always manage dynamic text. You might also find them useful for creating resizable application layouts. We discuss this more thoroughly in Chapter 9, but you've seen a few examples specifying component placement, height, and width, so what follows in Example 8-8 should make sense.

Example 8-8. *A dynamic, resizable layout involving bindings*

```
<s:Panel id="mainPanel" title="Main Panel" width="90%" height="90%">
    <s:Panel id="panel2" title="2nd Panel" x="10" y="10"
        height="{mainPanel.height * 0.25}" width="150"/>
    <s:Panel id="panel3" title="3rd Panel"
        x="10" y="{panel2.y + panel2.height + 10}"
        height="{mainPanel.height - panel2.height - 80}" width="150"/>
</s:Panel>
```

Example 8-8 binds the **height** and **y** values of two child panels to those of their parent, **mainPanel**. However, the bound **height** and **y** values undergo some math before the child panels can be rendered by Flash Player.

This example gets fairly busy. First, the second panel's **height** property is set to a quarter of the **height** for **mainPanel** with this code—`{mainPanel.height * 0.25}`. Next, the code `{panel2.y + panel2.height + 10}` sets the third panel's **y** value equal to the second panel's **y** value, plus its **height**, plus 10. The last 10 pixels give the two containers eye-friendly separation. Finally, the third panel's **height** is assigned by `{mainPanel.height - panel2.height - 80}`, which sets the **height** value equal to the **height** of the main panel, minus the **height** of the second panel, minus 80 pixels. The 80 pixels likely represent the

WARNING

Data bindings in layouts will prevent the layout from drawing correctly in Design mode. This happens because the component positioning is calculated by the Flash Player during runtime.

total space occupied by the three panels' title bars, plus an arbitrary number of pixels necessary to provide whitespace between the containers. Honestly, we found the 80 pixels using trial and error. Nothing fancy.

Because of the dynamic qualities of data binding, the overall result is a layout that resizes on the fly and maintains consistent positioning between containers. Figures 8-4 and 8-5 illustrate these squishy performance qualities.

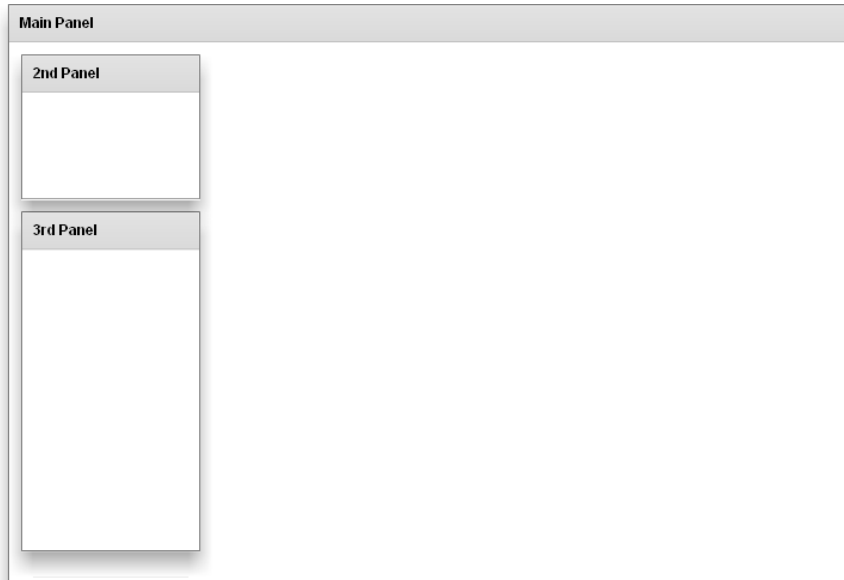


Figure 8-4. *The dynamic layout expanded*

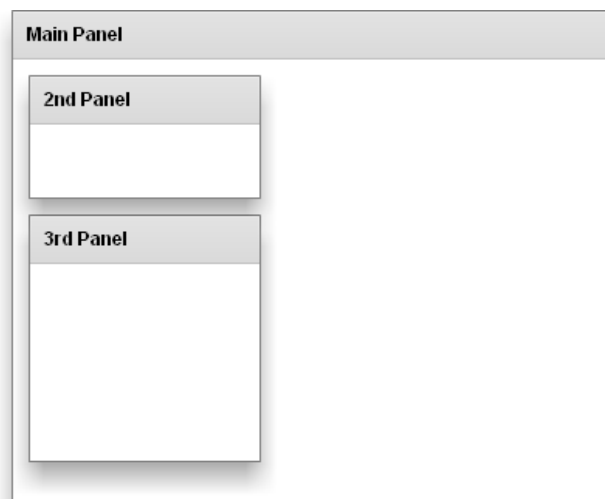


Figure 8-5. *The dynamic layout after shrinking the browser window*

The <fx:Binding/> Tag

In larger applications, you might find it useful to separate your binding declarations into a separate block of code. Our previous examples used curly braces and inline ActionScript to bind source values to their destinations. This is fine, and it's the typical way to establish bindings. However, you might find that inline bindings allow destinations to “know too much” about their sources. That's because, in the case of binding a **text** property to a variable, the attribute tag would refer to the variable explicitly.

Fortunately, there's a way to create data bindings using MXML tags, which we discuss next.

Basic usage

Example 8-9 demonstrates a binding using an MXML <fx:Binding/> tag.

Example 8-9. A data-binding relationship declared in MXML

```
<fx:Binding source="nameTI.text" destination="NameTI.text"/>

<s:VGroup horizontalAlign="center" horizontalCenter="0"
    verticalCenter="0">
    <s:Label text="Enter your full name.."/>
    <s:TextInput id="nameTI"/>
    <s:TextInput id="NameTI" editable="false"/>
</s:VGroup>
```

Run this, and it will look identical to Figure 8-1. In fact, both examples are functionally identical. Disregarding the fact that we're declaring the data binding in an MXML tag, there's only one difference between this example and Example 8-1. Can you see it?

The difference is this: the code for Example 8-1 did not require an **id** for the destination control, because the **destination** was implied by the inline binding. With the <fx:Binding/> tag, however, both controls must have a unique **id**. Example 8-9 further provided an opportunity to emphasize case-sensitivity in variable names by distinguishing the bound controls from one another using a single, strategic capital letter.

Some might say the <fx:Binding/> tag forces you to think more intently about the **source** and **destination** of a data-binding relationship, so this might represent an advantage that MXML bindings have over inline bindings.

Multiple sources

We discussed binding a source to multiple destinations in Example 8-3. However, a benefit of the <fx:Binding/> tag is the potential to specify *multiple sources* for a destination. For example, it's possible to bind a **Label** control's **text** to two or more sources. You accomplish this by declaring multiple <fx:Binding/> tags, as demonstrated in Example 8-10.

WARNING

Don't use braces within the source and destination properties of the <fx:Binding/> tag. For example, this is incorrect syntax:

```
<fx:Binding
    source="{nameTI.text}"
    destination="{label.text}"/>
```

NOTE

You can place `<fx:Binding/>` tags anywhere within the top level of your application, meaning you can place them between the opening and closing **Application** tags. A good practice is placing all your bindings in an area near the top of your application code.

Example 8-10. Multiple sources declared using MXML binding tags

```
<fx:Binding source="oneTextInput.text" destination="confusedLabel.text"/>
<fx:Binding source="anotherTextInput.text" destination="confusedLabel.text"/>

<s:VGroup horizontalCenter="0" verticalCenter="0">
    <s:TextInput id="oneTextInput"/>
    <s:TextInput id="anotherTextInput"/>
    <s:Label id="confusedLabel"/>
</s:VGroup>
```

In this code, the label's `text` property is bound to two different `TextInput` controls. If `oneTextInput` changes, its value is copied into the label. If `anotherTextInput` changes, its value is copied into the label.

It isn't necessary to use `<fx:Binding/>` tags for both binding declarations in this example. While curly braces alone can't designate multiple sources, curly braces can be used in conjunction with an `<fx:Binding/>` tag. However, using curly braces with an `<fx:Binding/>` tag can become confusing. Consider the code in Example 8-11, which has identical functionality to Example 8-10 but uses an `<fx:Binding/>` tag in conjunction with an inline binding to tether the label to two different `TextInput` controls.

Example 8-11. Combining an MXML binding with an inline binding to create a multisource binding relationship

```
<fx:Binding source="oneTextInput.text" destination="confusedLabel.text"/>

<s:VGroup horizontalCenter="0" verticalCenter="0">
    <s:TextInput id="oneTextInput"/>
    <s:TextInput id="anotherTextInput"/>
    <s:Label id="confusedLabel" text="{anotherTextInput.text}"/>
</s:VGroup>
```

WARNING

You cannot place an `<fx:Binding/>` tag within a container.

Curly Brace Syntax Versus Binding Tags

The `<fx:Binding/>` tag offers essentially the same functionality as curly braces, just a different way of approaching it. With the `<fx:Binding/>` tag, you can organize your bindings together in a central area of the code. Plus, if you have a situation requiring such functionality, you can bind a single destination to multiple sources. On the other hand, because you're limited to one source and one destination, you can't create intricate bindings like you can with multiple sets of curly braces.

Two-Way Bindings

Flex 4 introduced a new ability to create *two-way bindings*, sometimes referred to as *bidirectional bindings*. Two-way bindings allow either side of the binding expression to act as the source or the destination. When one side changes, the other side absorbs the change. You might wonder why this doesn't trigger an infinite loop, but these two-way binding methods only consider changes triggered through the user interface.

In the previous version of Flex, bidirectional binding required writing two separate binding expressions that traded source and destination pairs. However, Flex 4 offers native two-way data binding. Example 8-12 demonstrates two-way data binding using both an `<fx:Binding/>` tag and inline script.

Example 8-12. *Creating two-way data bindings using both the MXML tag and the shorthand inline syntax*

```
<fx:Binding source="firstTI.text" destination="secondTI.text"
    twoWay="true" />

<s:VGroup horizontalCenter="0" verticalCenter="0">
    <s:Label text="Bound using a two-way MXML binding tag.." />
    <s:TextInput id="firstTI" />
    <s:TextInput id="secondTI" />
    <s:Label text="Bound using shorthand, two-way binding syntax.." />
    <s:TextInput id="thirdTI" text="@{fourthTI.text}" />
    <s:TextInput id="fourthTI" />
</s:VGroup>
```

To begin with, notice the `twoWay` attribute attached to the `<fx:Binding/>` tag. With the `twoWay` property set to `true`, an `<fx:Binding/>` tag creates a bidirectional binding relationship between its `source` and `destination` values.

The second pair of `TextInput` controls demonstrates two-way binding using shorthand syntax. The at symbol (@) preceding the braces indicates a bidirectional binding between the two participating components.

If you test that code, you'll discover both implementations of two-way binding yield identical results. It's mostly a question of which syntax you prefer—`<fx:Binding/>` tags or inline script.

Yet Another Way to Bind

There's yet another way to create bindings in Flex. With the `BindingUtils` class, you can create your own bindings in ActionScript. Why would you want to do this? Well, if you need fine-grained control over when bindings fire, or if you need to turn bindings off and on at will, ActionScript affords you more control. If the `<fx:Binding/>` tag and the curly brace syntax don't suit your needs, peruse the Language Reference for the `mx.binding.utils.BindingUtils` class.

This is an advanced approach, and therefore it's outside the scope of this book, but we wanted you to be aware of the additional potential.

Handling Complex Data with Data Models

A *data model* is a single object with multiple properties you can use to store a lot of related information in one place. Flex makes it easy to store structured information in data models, and as we will see, data binding gives us a useful mechanism to benefit from this flexible component.

Building the Data Model

Data models can help you organize your code, and they can be quite practical. For instance, when you're pulling data from a server, it's good practice to grab lumps of similar data, such as a person's name, email, and address, and then store the result in a model for later use.

Example 8-13 demonstrates the data model. Do you see something new? The data model must be declared within a `<fx:Declarations/>` block, which we've been prone to deleting since we haven't been using it.

Example 8-13. A data model for storing user information

```
<fx:Declarations>
  <fx:Model id="contact">
    <info>
      <name>
        <first>John</first>
        <last>Doe</last>
        <full>{contact.name.first} {contact.name.last}</full>
      </name>
      <email>john.doe@foo.com</email>
      <phone>555-555-555</phone>
    </info>
  </fx:Model>
</fx:Declarations>
```

NOTE

Note that you use the expression **contact.phone** and not **contact.info.phone** to get the phone number. That's because the **id** refers to the root of the data model.

This model allows us to store user information in a central location that's easy to read and understand. Notice the root of the XML structure is a tag called `<info/>`. A root tag must be included to create valid XML, but you can call the root anything you want; the word "info" isn't a requirement.

The beauty of a data model is that it accesses the model's information elsewhere in your application using dot notation. For instance, to access the phone number from the model in the previous example, you'd use the expression **contact.phone**. Unfortunately, although you can access the model's data using dot notation, code completion won't recognize the tags used in the model, so you'll need to make a point to remember the model's structure.

Multilevel Bindings

Example 8-13 established a data model and populated it with some default information. However, notice the following code inside the `<name/>` tag:

```
<full>{contact.name.first} {contact.name.last}</full>
```

The model builds the contact's full name by binding and concatenating the **first** and **last** name properties available within its own content.

Also, what about our original notion that this customer data was poured in from a server? Well, with two-way binding, information in the model can not only originate at the server, but it can be resubmitted to the server when its values change. The mixture of these binding relationships constitutes *multilevel binding*.

Although we can't simulate the server side of this example, we can demonstrate the overall interaction of data entering and then exiting the model. To demonstrate multilevel binding, set the data model from Example 8-13 into an application along with the MXML in Example 8-14. Figure 8-6 illustrates this process.

Example 8-14. *Multilevel binding to a model*

```

<s:HGroup horizontalCenter="0" verticalCenter="0" verticalAlign="middle">
  <s:VGroup horizontalAlign="right">
    <mx:FormItem label="First Name:">
      <s:TextInput id="firstNameTI" text="@{contact.name.first}"/>
    </mx:FormItem>
    <mx:FormItem label="Last Name:">
      <s:TextInput id="lastNameTI" text="@{contact.name.last}"/>
    </mx:FormItem>
    <mx:FormItem label="Email:">
      <s:TextInput id="emailTI" text="@{contact.email}"/>
    </mx:FormItem>
    <mx:FormItem label="Phone:">
      <s:TextInput id="phoneTI" text="@{contact.phone}"/>
    </mx:FormItem>
  </s:VGroup>
  <s:VGroup paddingLeft="75">
    <s:Label text="Confirmation" fontWeight="bold"/>
    <s:Label text="{contact.name.full}"/>
    <s:Label text="{contact.email}"/>
    <s:Label text="{contact.phone}"/>
  </s:VGroup>
</s:HGroup>

```

First Name:	<input type="text" value="Elijah"/>	Confirmation Elijah Doe john.doe@foo.com 555-555-555
Last Name:	<input type="text" value="Doe"/>	
Email:	<input type="text" value="john.doe@foo.com"/>	
Phone:	<input type="text" value="555-555-555"/>	

Figure 8-6. *Data goes in and data goes out in one fluid motion*

This code contains two sets of components. Components in the first group perform two-way data binding against the model and its properties. Meanwhile, the second group reads from the model via standard one-way bindings.

When the application is launched, both groups will populate with the model's default values. However, as changes are entered into the first group's controls, the model will update and immediately bind the changed data to the second group's controls. This is multilevel binding. Perhaps you can imagine how multilevel binding can save us from writing a lot of code to accomplish the same result.

You can use all the features of inline binding within data models, making data coercion easy. In other words, you can take advantage of models to set up string concatenation as well as other *data-massaging* tasks, meaning piecing together, running calculations on, and possibly formatting data for display.

NOTE

Perhaps you noticed the property **paddingLeft** adding whitespace between the two **VGroup** containers? We've sprinkled layout tricks into this chapter's examples to get you excited about Chapter 9, where we cover layouts in greater detail.

NOTE

If you need to perform advanced data management, you might find a class-based model more efficient. Refer to the *Flex documentation* about class-based models for more information on this topic.

When Data Binding Isn't Appropriate

Although it's powerful and easy, data binding isn't always the best solution. If your application relies on a timing mechanism to display data, data binding wouldn't be appropriate, because you wouldn't have control over when the binding is triggered.

Bindings can also create issues when they're rapidly and repeatedly firing. For instance, a binding fires whenever its source value changes, so several properties relying on a dense and frequently updated source may fire bindings too often, hurting performance in the process.

In these cases, it may be wiser to handle your data assignments in ActionScript.

Summary

This chapter introduced you to data binding in Flex. You learned how to apply bindings for several situations, and you should be comfortable using bindings in the following contexts:

- Declaring inline, component-to-component bindings within MXML
- Creating `[Bindable]` variables inside a Script/CDATA block
- Concatenating text inline using binding values
- Defining dynamic, resizable application layouts
- Declaring binding relationships using the `<fx:Binding/>` tag
- Building two-way bindings using either the shorthand, inline syntax or the `twoWay="true"` property of a `<fx:Binding/>` tag
- Assembling multilevel bindings between a data model and various components

Data binding can be an invaluable tool for moving information around in your applications, and now you're familiar with several approaches to data binding. Since data binding is used so frequently in Flex applications, you will continue to use and expand on this important concept throughout the book. For instance, data binding comes up again when we discuss the *dataProvider* property of **List**-based components, which is slated for Chapter 11.

In the next chapter we delve into the details of Flex application layouts.

DESIGNING APPLICATION LAYOUTS

“The Chinese workingmen, hanging in their baskets, had to bore the holes with their small hand-drills, then tamp in the explosives, set and light the fuse, and holler to be pulled out of the way.”

—Stephen Ambrose,
Nothing Like It in the World

Fortunately for us, creating a Flex application does not involve life-threatening hazards, but like with any construction project, some planning is necessary.

Engineered projects can be judged by their structural integrity—the wall is plumb, the bridge supports, the dam holds. Similarly, your applications stand to gain from effective planning—the design is lightweight, the content is well arranged, the interface is intuitive.

One situation brought upon by today’s multitude of screen resolutions is the need to develop a layout that can expand and contract. If someone is viewing an application in a web browser, she can resize her browser. How will your layout react? You should anticipate and make an effort to accommodate this scenario.

Flex development provides a number of approaches for arranging Flex applications. You can use precise coordinate positioning, or you can use relative positioning (i.e., vertical/horizontal). You can also use constraints to force components into positions relative to a container’s center or its edges. Most often, though, you’ll combine all three approaches in order to obtain the perfect layout.

In this chapter, we help you gain an understanding of layout design by covering the types of layouts, layout containers and their common properties, and tactics for developing dynamic, resizable layouts using bindings and constraints.

IN THIS CHAPTER

- Types of Layouts
- The Display List
- Sizing
- Controlling Whitespace in the Layout
- Advanced Containers
- Spacers and Lines
- Alignment
- Constraints-Based Layout
- Summary

Types of Layouts

Spark and Halo containers honor the same layout styles: Absolute/Basic, Horizontal, and Vertical. However, layouts for Spark containers are configured differently than layouts for Halo containers. To make the distinction, Halo containers take layout styles as property assignments, like in Example 9-1.

Example 9-1. A Halo Panel container configured for absolute layout

```
<mx:Panel id="haloPanel" layout="absolute">
  <!-- components go here -->
</mx:Panel>
```

Alternatively, Spark containers require nested layout declarations, like in Example 9-2.

Example 9-2. A Spark Panel container configured for basic layout

```
<s:Panel id="sparkPanel">
  <s:layout>
    <s:BasicLayout/>
  </s:layout>
  <!-- components go here -->
</s:Panel>
```

Now that you know how layouts are assigned to containers in both component sets, next we discuss the absolute and relative layouts, as well as the Spark containers defaulting to these types.

"Pay as You Go"

In comparison with the Halo components, Spark components have been optimized to reduce compiled SWF size. The concept motivating the transition from Halo to Spark components has been labeled "pay as you go." Let's consider this concept with regard to containers.

Basically, Halo components offer so many properties and styles that Flex 3 applications are often encumbered by a surplus of these, at a cost of increased file size. For example, it's easy to create a Halo **HBox** container, give it some components, and, if desired, declare border qualities such as **borderStyle**, **borderColor**, **cornerRadius**, etc. However, if you don't assign the border properties, you still end up compiling that container with all its overhead into your application.

Compared to Halo containers, Spark containers are stripped down. For instance, Spark's **HGroup** container does not support border or background properties. If you need these properties, you should use a Spark **BorderContainer** with a Horizontal layout. Because you have the liberty to choose the most optimal component for the situation, a Flex 4 application with several **HGroup** containers stands to be much *leaner* than its Flex 3 counterpart with several **HBox** containers. There is one downside to this new arrangement: Flex 4 projects require more lines of code to create the same layout.

Because this underlying difference has the potential to create much smaller Flex applications, Adobe recommends using components from the Spark package whenever possible.

Absolute Positioning

When you create a new Flex application, the **Application** container defaults to an *absolute layout*. This means components are positioned within the **Application** container using **x** and **y** coordinate assignments. Absolute positioning is best suited for developing small widgets and applets, as these types of applications usually need to fit a well-defined, fixed portion of a larger website, making it important to control their exact dimensions.

The term “absolute” is a holdover from Halo terminology, but it is synonymous with **BasicLayout** in Spark terminology. An advantage of this layout scheme is the freedom to stipulate exactly where you want to locate components within their parent containers. The absolute layout also has some disadvantages. Namely, absolute layouts do not dynamically resize. Also, because components are positioned independently and without regard for one another, they may overlap.

In addition to the **Application** container, the following Spark containers also support the Absolute/Basic layout:

<s:Group/>

The Spark **Group** container arranges its children using absolute positioning. It doesn't offer background or border styling properties, and it's not skinnable. This is one of the most lightweight containers in the Flex framework.

<s:BorderContainer/>

This **BorderContainer** is appropriate for layouts/containers requiring background and border styling. It defaults to a Basic layout, but it also recognizes nested Horizontal and Vertical layouts.

<s:SkinnableContainer/>

The **SkinnableContainer** is an advanced container that should be used when you want to apply a special skin to a Spark container.

As you realize, the **BasicLayout** isn't your only option. Containers that use relative positioning can arrange their child components into either a Horizontal or a Vertical layout.

Relative Positioning

Containers using a *relative layout* align their components either horizontally or vertically. The **HGroup** and **VGroup** impose Horizontal and Vertical layouts, respectively, as their default behavior without the need for nested layout declarations.

Both the **HGroup** and the **VGroup** are *convenience containers*; their layout declaration is built-in. In other words, they can be re-created manually by pairing a **Group** with a nested layout declaration. Example 9-3 demonstrates two sets of buttons in vertical alignment. The first set uses a **VGroup**, and the second set uses a **Group** with a **VerticalLayout** declaration; see Figure 9-1. This example emphasizes similarity between the **HGroup/VGroup** and other containers using nested Horizontal/Vertical layouts. Note that the **Application** container uses a nested **HorizontalLayout**, and although properties such as **horizontalAlign** can be applied directly to the **HGroup/VGroup**, they should be assigned directly to the layout when using a nested layout declaration.

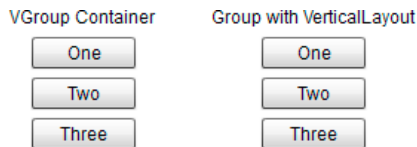


Figure 9-1. On the left, buttons aligned vertically using a **VGroup**, and on the right, a **Group** container with a nested **VerticalLayout** declaration

Example 9-3. *VGroup versus Group and VerticalLayout*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:HorizontalLayout gap="25"
      horizontalAlign="center"
      verticalAlign="middle"/>
  </s:layout>

  <s:VGroup horizontalAlign="center">
    <s:Label text="VGroup Container"/>
    <s:Button label="One"/>
    <s:Button label="Two"/>
    <s:Button label="Three"/>
  </s:VGroup>

  <s:Group>
    <s:layout>
      <s:VerticalLayout horizontalAlign="center"/>
    </s:layout>
    <s:Label text="Group with VerticalLayout"/>
    <s:Button label="One"/>
    <s:Button label="Two"/>
    <s:Button label="Three"/>
  </s:Group>

</s:Application>
```

Unlike the **HGroup** and the **VGroup**, which are specialized convenience containers, the **Application**, **Group**, **BorderContainer**, **SkinableContainer**, and **Panel** containers all support nested layouts. For this reason, they may be called *hybrid containers*.

The next section examines another important topic relevant to application layout and component containment: the Display List.

The Display List

The *Display List* refers to the hierarchy of containment and layering describing every graphical element drawn for an application. Containment begins with the **Application** root, and it extends through each nested container and each container's nested elements. However, the Display List is not limited to the obvious. For instance, even a **Button** contains multiple graphical elements, such as its **Label** and the graphics that make it look like a button. Basically, think of the Display List in terms of containers and components.

Default Layer Order

If you're familiar with design programs, you know that more recently added layers overlap older ones. This is similar to how MXML code translates into an application. Elements declared toward the bottom of your MXML code (i.e., last) will be drawn closer to the surface/top of the resulting application.

The layering relationship between graphical components is managed using an *index* value. In programming jargon, the first component added has the lowest index, and the index value increases by one for every additional component. In Example 9-4, as we add **BorderContainer** components to a **Panel**, each subsequent element will overlap the previous, and the last element will have the highest index.

Example 9-4. The last graphic (blueBox) has the highest index in the Display List

```
<s:Panel id="colorsPanel"
  title="Demonstrating the Display List"
  width="250" height="250">

  <s:layout>
    <s:BasicLayout/>
  </s:layout>

  <s:BorderContainer id="redBox" x="70" y="70" height="50" width="50"
    backgroundColor="#FF0000">
  </s:BorderContainer>

  <s:BorderContainer id="greenBox" x="90" y="90" height="50"
    width="50" backgroundColor="#00FF00">
  </s:BorderContainer>

  <s:BorderContainer id="blueBox" x="110" y="60" height="50"
    width="50" backgroundColor="#0000FF">
  </s:BorderContainer>

</s:Panel>
```

This code creates a Panel with three swatches: red, green, and blue. The red element, which is the first to be declared in the MXML, is the first to be rendered at runtime. It gets the lowest index and is ultimately overlapped by both the green and the blue elements. The blue swatch, however, is the last to be declared in the MXML, is rendered last, and has the highest index. See for yourself in Figure 9-2.

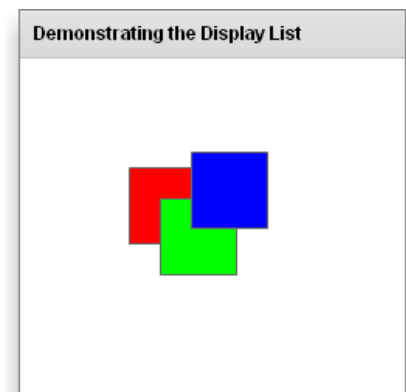


Figure 9-2. Overlapping swatches illustrating containment order

Accessing Children

Considering how layers are ordered in the Display List, it's possible to programmatically access and modify the components in your application. We'll demonstrate this by modifying the layering order of the color swatches in the previous example.

For this exercise, you'll make use of two methods: `getElementIndex()`, which returns the index position of an object in the Display List, and `setElementIndex()`, which changes the index position of an object in the Display List. These methods are available only to containers. To use these methods, you first reference the **id** of a container holding an element you want to access; then, you call one of the former methods and pass it an instance of the element.

Think for a moment about Example 9-4. The following code would return the index value of the component having the **id** of `blueBox` that is contained by `colorsPanel`:

```
colorsPanel.getElementIndex(blueBox);
```

Since index values are zero-based (meaning they start at 0 and increment from there), this line would return a value of 2. Similarly, the red swatch would return 0, and the green element would return 1.

The method `setElementIndex()` is similar, but its purpose is to change an object's position in the Display List. This method takes a reference to a component in the Display List as well as an **int** value, which it uses to reassign the component's position:

```
colorsPanel.setElementIndex(blueBox,0);
```

This line, taking a reference to `blueBox` and the supplied index of 0, moves the `blueBox` element from its current position, 3, into index position 0. It has the effect of moving `blueBox` behind the other color swatches.

Knowing all this, how can we programmatically change the index position of `blueBox`? Since we'll need an event to call `setElementIndex()`, we'll add a **Button** to Example 9-4 to call the function, `moveBlueBox()`, which will handle the index reassignment.

Create the application in Example 9-5, called **MoveBlueBox**, and after you compile it, notice how `blueBox` moves back and forth in the Display List as you click the Move button (Figure 9-3). The next section will continue using this example, so be sure to keep it around.

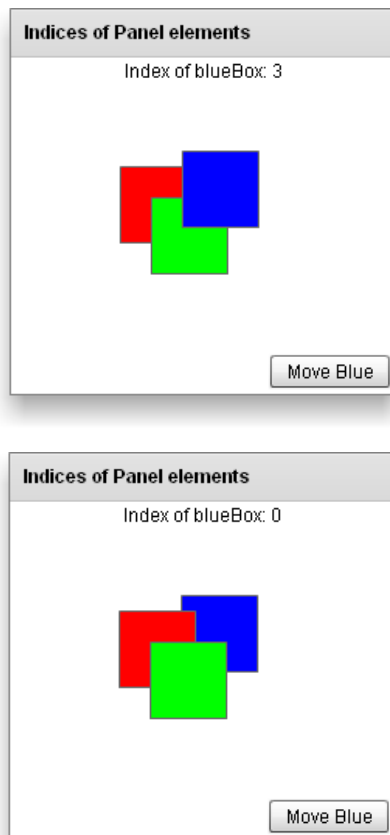


Figure 9-3. Overlapping swatches illustrating containment order

Example 9-5. Moving *blueBox* back and forth between layer positions 3 and 0

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="updateLabel()">

  <fx:Script>
    <![CDATA[
      private function updateLabel():void{
        var blueIndex:int =
          colorsPanel.getElementIndex(blueBox);
        label.text = "Index of blueBox: " + String(blueIndex);
      }

      // decide where to move blueBox, then update label
      private function moveBlueBox():void{
        if(colorsPanel.getElementIndex(blueBox) == 3){
          colorsPanel.setElementIndex(blueBox,0);
        }else{
          colorsPanel.setElementIndex(blueBox,3);
        }
        updateLabel();
      }
    ]]>
  </fx:Script>

  <s:Panel id="colorsPanel" title="Indices of Panel elements"
    width="250" height="250" horizontalCenter="0" verticalCenter="0">

    <s:layout>
      <s:BasicLayout/>
    </s:layout>

    <s:Label id="label" y="4" horizontalCenter="0"/>

    <s:BorderContainer id="redBox" x="70" y="70" height="50"
      width="50" backgroundColor="#FF0000">
    </s:BorderContainer>

    <s:BorderContainer id="greenBox" x="90" y="90" height="50"
      width="50" backgroundColor="#00FF00">
    </s:BorderContainer>

    <s:BorderContainer id="blueBox" x="110" y="60" height="50"
      width="50" backgroundColor="#0000FF">
    </s:BorderContainer>

    <s:Button id="moveButton" right="4" bottom="4"
      label="Move Blue" click="moveBlueBox()"/>

  </s:Panel>
</s:Application>

```

NOTE

The equality operator (`==`) in the **If..Else** block is handling the decision of whether **blueBox** should be moved from position 3 to 0, and vice versa. If the current position of **blueBox** is 3, then we'll set it to 0; otherwise, we'll return **blueBox** back to display position 3.

NOTE

The **moveButton** is positioned using constraints—**right="4"** and **bottom="4"**. We discuss constraints later in this chapter.

Adding and Removing Children

Sometimes it's beneficial to add and remove components using ActionScript. To do this, use dot notation to reference a container and call its `addElement()` or `removeElement()` methods. To demonstrate, we'll add a few lines of code to Example 9-5 that allow us to toggle a yellow swatch. Start by adding a `CheckBox` component below the `Button`, as shown in Example 9-6.

Example 9-6. A `CheckBox` to toggle component addition/removal

```
<s:CheckBox id="toggleCheckBox" left="4" bottom="4" label="Toggle Yellow"
  change="toggleYellowBox()"/>
```

As you can see, we'll handle the toggling with a function called `toggleYellowBox()`, but first, add the variable declaration from Example 9-7 at the top of a `Script` block.

Example 9-7. A class-level variable to hold an instance of a component

```
<![CDATA[
  private var yellowBox:BorderContainer;
```

Finally, add the function `toggleYellowBox()`, shown in Example 9-8, just below `moveBlueBox()`. The function uses the `CheckBox` control's `selected` property to decide whether to add or remove the `yellowBox`.

Style Smarts

You won't find `"backgroundColor"` among the code completion options when assigning properties to an instance of the `BorderContainer` class. In ActionScript, style properties are defined using the `setStyle()` method, which requires you to provide the style name as a string and an *appropriate value* for the style property.

Considering our example, we're passing the style `backgroundColor` and the property `0xFFFF00`, which is the hex value for yellow:

```
yellowBox.setStyle("backgroundColor", 0xFFFF00);
```

We could get the same result by passing the color value as a string:

```
yellowBox.setStyle("backgroundColor", "#FFFF00");
```

And here's another option for passing the color as a string:

```
yellowBox.setStyle("backgroundColor", "yellow");
```

So, each of these approaches to style assignment will work; however, passing the hex value might have a slight performance advantage.

You may not always remember which styles a component supports or what data types those styles can accept. If you're in this situation, use Adobe's Language Reference or Design mode's Properties pane to determine the styles available to a component. Alternatively, if you don't want to leave Source mode, you can "borrow" code completion on an MXML component for hints, but if you don't have an existing component to work with, you'll have to make an arbitrary MXML component just to peruse its styles. To some (like us), this is preferable to searching the Web or clicking around Flash Builder, so we felt this tip was worth sharing.

Example 9-8. The function to add or remove yellowBox

```
private function toggleYellowBox():void{
    if (toggleCheckBox.selected){
        yellowBox = new BorderContainer();
        colorsPanel.addElement(yellowBox);
        yellowBox.id = "yellowBox";
        yellowBox.x = 130;
        yellowBox.y = 75;
        yellowBox.height = 50;
        yellowBox.width = 50;
        yellowBox.setStyle("backgroundColor", 0xFFFF00);
    }else{
        colorsPanel.removeElement(yellowBox);
        yellowBox = null;
    }
}
```

This function should be rather straightforward. The **CheckBox** control's **change** event calls the function, and the control's **selected** property determines whether a yellow swatch is added to or removed from the display (Figure 9-4).

When the component is added, the **new** keyword creates an instance of the **BorderContainer** class, which is then added to the container using its **addElement()** method. The component's properties are defined using dot notation; however, note the **setStyle()** method defining the **backgroundColor** property. In **ActionScript**, you use **setStyle()** to assign style properties, because they aren't exposed or accessible using dot notation. For more information concerning the **setStyle()** method, see the sidebar titled "Style Smarts" on page 158.

The second half of the function eliminates **yellowBox** by first removing it from the **Panel** using the **removeElement()** method. Last, we assign **yellowBox** a **null** value to allow full destruction of its in-memory representation by a process known as *garbage collection*. For more information on Flash Player's garbage collection routine, see the sidebar below, titled "Take Out the Trash."

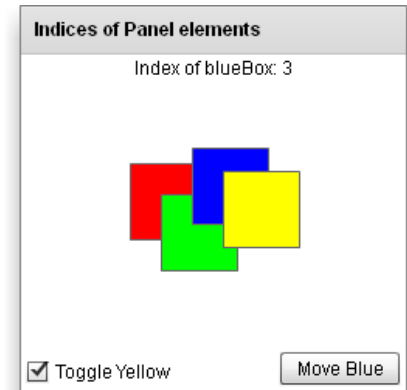


Figure 9-4. Toggling a yellow swatch using **ActionScript**

Take Out the Trash

In Example 9-8, we create a new instance of **yellowBox** each time we add it; therefore, we should set the current instance of **yellowBox** to **null** each time we remove it. This practice ensures we fully destroy our instance of the component, which keeps the system memory free of waste. The process handling memory cleanup is called *garbage collection*, and it occurs behind the scenes in Flash Player.

Consider this comparison: the sanitation workers who dispose of our trash require us to first remove the trash from our house. This is the same concept we're applying in code; setting **yellowBox** to **null** effectively takes our application's trash to the curb so Flash Player's garbage collector can destroy it in memory.

Rearranging Children

Besides what you saw previously, there are other methods available for rearranging components. The container methods `swapElements()` and `swapElementsAt()` allow you to literally swap display positions between a couple elements in a container. They perform identically, except that the first method takes component references whereas the second takes index values. To see for yourself, add the line of code in Example 9-9 at the bottom of the true condition for `toggleYellowBox()`.

Example 9-9. *This line swaps display positions of `blueBox` and `yellowBox`*

```
colorsPanel.swapElements(blueBox, yellowBox);
```

Moreover, just as you can manage the layer depth of elements in a container, you can also move elements between containers. This means the colored boxes aren't limited to movement within the `Panel`; rather, they can be moved anywhere, such as to another container or the `Application` itself. To demonstrate, the lines of code in Example 9-10 would move the `Label` control from `colorsPanel` to `yellowBox`.

Example 9-10. *Moving the `Label` from one container to another*

```
colorsPanel.removeElement(label);
yellowBox.addElement(label);
```

Sizing

There's more to controlling component size than specifying pixel dimensions. While containers can provide arrangement, in some situations they will resize themselves to fit their children, or resize their children to fit themselves.

Explicit Sizing

The *explicit sizes* refer to a component's **height** and **width** properties. Although many containers and controls have a reasonable default size, sometimes they expand to accommodate the components they contain.

You should know, however, that Spark and Halo containers behave differently when their specified dimensions are less than those of their content. Spark containers require more particular accounting; in other words, the `HGroup` and `VGroup` will ignore explicit dimensions and expand to fit their content. On the other hand, when a Halo container is short of interior space, it will clip its content and create scroll bars.

Examples 9-11 and 9-12 and Figures 9-5 and 9-6 demonstrate these behaviors for an `HGroup` and an `HBox`, respectively. Both containers are given an explicit **width** of 85 pixels, yet their children require a combined space of 100 pixels; notice how each container handles the mismatch.

NOTE

Some controls resize themselves based on their properties. For instance, both `Checkbox` and `Button` will expand to fit long labels.

Example 9-11. A Spark HGroup with oversized contents

```
<s:HGroup width="85" horizontalCenter="0" verticalCenter="0">
  <s:BorderContainer id="greenBox" height="50" width="50"
    backgroundColor="#00FF00">
  </s:BorderContainer>

  <s:BorderContainer id="blueBox" height="50" width="50"
    backgroundColor="#0000FF">
  </s:BorderContainer>
</s:HGroup>
```

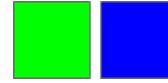


Figure 9-5. An HGroup resizing to accommodate its children

Example 9-12. A Halo HBox with oversized contents

```
<mx:HBox width="85" horizontalCenter="0" verticalCenter="0">
  <s:BorderContainer id="greenBox" height="50" width="50"
    backgroundColor="#00FF00">
  </s:BorderContainer>

  <s:BorderContainer id="blueBox" height="50" width="50"
    backgroundColor="#0000FF">
  </s:BorderContainer>
</mx:HBox>
```

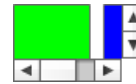


Figure 9-6. A “rule-crazy” HBox maintaining explicit width

Relative or Percentage-Based Sizing

Sometimes you want to size components and containers according to their relationship with one another; this is called *relative sizing* or *percentage-based sizing*. For example, a **Button** can have an explicit **width** of 22 pixels; alternatively, it can have a relative width, such as 50%. The relative width requires the **Button** to dynamically resize itself to consume 50% of its parent container’s width.

The code in Example 9-13 creates a 400-pixel-wide **Group** with a **Button** child set to a relative width of 50% (see Figure 9-7).

Example 9-13. A Button with a relative width assignment

```
<s:Group width="400" height="100">
  <s:Button label="Button" width="50%"/>
</s:Group>
```

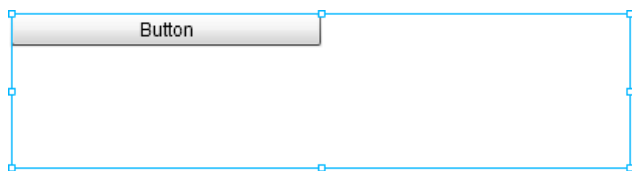


Figure 9-7. A Button with percentage-based width consuming 50% of its container

WARNING

Components will compete for available space when using percentage-based sizes. If you’re using several percentage-based sizes and getting unexpected results, that might explain why.

NOTE

MXML allows you to set a component’s **height** and **width** properties using either exact integer values or percentages. However, to set percentage dimensions in ActionScript, use the properties **percentHeight** and **percentWidth**, which accept values in the range 0–100 and correspond to percentage values.

Because its container is 400 pixels wide, the button will consume half of that, or 200 pixels. You could also set the **Group** to a relative size, say, 100%. This would cause the **Group** to consume all the available space of its container, and then the **Button** would take half of that. In the latter case, if the application were resized, the widths of **Group** and **Button** would also change.

Minimum and Maximum Sizes

You also have control over minimum and maximum heights and widths. This is especially useful when used in concert with percentage-based sizes.

For example, if you wanted an application to occupy 100% of a browser window, thus allowing it to resize along with the browser, you could use **minHeight** and **minWidth** to ensure your application maintains some minimum dimensions, such as 800×600 or 1024×768, in order to look and perform as you designed it (Example 9-14). Without set minimums, a tightly resized browser might create trouble for your layout.

Example 9-14. *An application designed to occupy 100% of the browser while maintaining a minimum size of 800×600 if the browser is reduced*

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  height="100%" minHeight="600" width="100%" minWidth="800">
```

Controlling Whitespace in the Layout

The relative layouts (Horizontal/Vertical) provide a few options to help control whitespace between their child components. This section explores paddings and gaps, which are the properties used to control whitespace in relative layouts.

Padding the Layout

If you've used much CSS, you're probably familiar with the concept of padding, which lets you specify a number of pixels by which to pad a container. *Padding*, which is available only to the relative layouts, creates space between a container's borders and its child components. The specific padding properties include **paddingLeft**, **paddingRight**, **paddingTop**, and **paddingBottom**.

If you add a few buttons to a **Panel** with a vertical layout, by default the buttons will situate at the top-left corner, align vertically, and hug the left edge of the container. However, you can create space between a container and its components by setting padding properties, as demonstrated by Examples 9-15 and 9-16 and Figure 9-8.

In Spark containers, padding is applied as properties of the nested layout, like in Example 9-15.

Example 9-15. Applying padding to the nested layout of a Spark Panel

```
<s:Panel title="Padding">
  <s:layout>
    <s:VerticalLayout paddingTop="10" paddingLeft="10"/>
  </s:layout>
  <s:Button label="Button"/>
  <s:Button label="Button"/>
  <s:Button label="Button"/>
</s:Panel>
```

Halo containers, on the other hand, take padding properties directly; notice also in Example 9-16 that the layout is declared as a container attribute.

Example 9-16. Applying padding to the built-in layout of a Halo Panel

```
<mx:Panel title="Padding" height="125" width="125"
  layout="vertical" paddingTop="10" paddingLeft="10">
  <s:Button label="Button"/>
  <s:Button label="Button"/>
  <s:Button label="Button"/>
</mx:Panel>
```

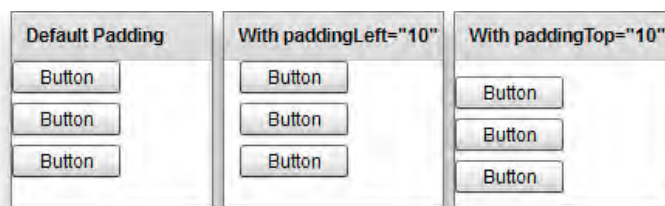


Figure 9-8. Panels applying different padding properties

Gaps

Relative layouts also support gaps. *Gaps* refer to the amount of whitespace between nested content elements.

Like padding, gaps are configured differently between Spark and Halo containers; moreover, the property names are different. Spark layouts use the more generic property **gap**, and it's applied directly to the nested layout. On the other hand, Halo containers recognize either **horizontalGap** or **verticalGap**, and the setting is applied directly to the container's attributes. Examples 9-17 and 9-18 and Figure 9-10 demonstrate the application of gaps to both Spark and Halo containers.

The Flex Layout Process

The *Flex layout process* passes through three phases and considers several variables.

First, during the *commitment pass*, components such as **Buttons** and **CheckBoxes** that resize to accommodate internal elements are given the chance to do so. Next, starting with the most deeply nested components, the *measurement pass* collects explicit, pixel-based sizes. Percentage-based sizes aren't considered yet. Finally, during the *update pass*, the renderer applies size and placement in reverse order. Because fixed sizes were considered during measurement, percentage-based sizes can now be calculated.

Next we walk through this process and consider the following code. Just for clarification, the **Button** is the most deeply nested component, followed by the **Panel**, then the **Group**, and finally the **HGroup**:

```
<s:HGroup>
  <s:Group width="300">
    <s:Panel width="100%">
      <s:Button label="One long button label"/>
    </s:Panel>
  </s:Group>
</s:HGroup>
```

First, during the commitment pass, the **Button** resizes to fit its unusually long **label** property. Next, starting with the **Button** and moving toward the root, explicit sizes are measured. In this case, only the **Button** and the **Group** have known sizes. With fixed sizes known, the update pass can place the elements. First, the **HGroup** is added to its parent, and it sizes to accommodate its nested, 300-pixel-wide **Group**. Then, the **Group** is added. Next, because the **Panel** has a **width** of 100%, it's sized to fill the **Group**. Finally, the **Button** is added to the **Panel**. See Figure 9-9 for an illustration of this process.

While an application is in use, any changes to component placement or resizing of the application window will cause the entire layout process to repeat. By walking through this process, we're making a point for efficient use of containers. As you can see, every time an application is redrawn, Flash Player does a lot of work.

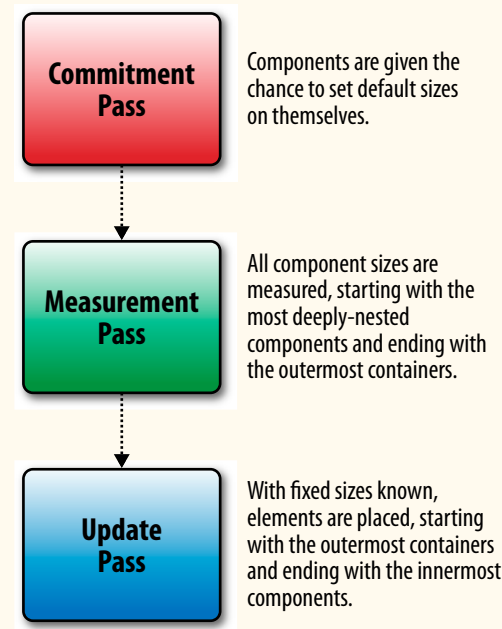


Figure 9-9. The three steps in the Flex layout process

The code in Example 9-17 applies a **gap** of 20 pixels to a Spark **Panel** with a **VerticalLayout**.

Example 9-17. Applying a gap to the nested layout of a Spark Panel

```
<s:Panel title="Gaps" height="156" width="175">
  <s:layout>
    <s:VerticalLayout gap="20"/>
  </s:layout>
  <s:Button label="Button"/>
  <s:Button label="Button"/>
  <s:Button label="Button"/>
</s:Panel>
```

And the code in Example 9-18 demonstrates the same for a Halo **Panel** with a vertical **layout** property.

Example 9-18. *Applying a gap to the built-in layout of a Halo Panel*

```
<mx:Panel title="Gaps" height="156" width="175"
  layout="vertical" verticalGap="20">
  <s:Button label="Button"/>
  <s:Button label="Button"/>
  <s:Button label="Button"/>
</mx:Panel>
```

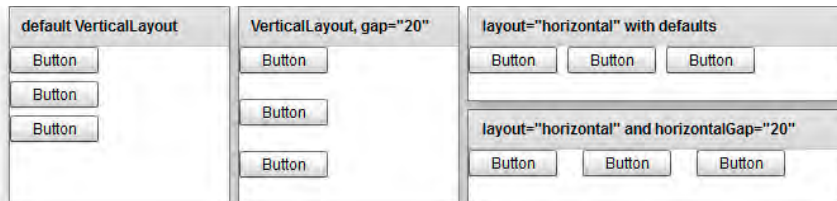


Figure 9-10. *Panels with different layouts and gap styles*

Advanced Containers

A few containers provide advanced functionality for situations that require a more complicated layout.

Divided Boxes

The Halo *Divided Box* containers are unique in that they provide resizable content areas; however, these containers are similar to their group-based cousins, **HGroup** and **VGroup**, in that they use relative layouts to arrange their children horizontally or vertically.

The Divided Box creates a draggable boundary between each of its children, which are frequently other containers. Of course, moving the draggable boundary resizes the adjacent content areas. Divided boxes make good use of percentage-based layouts in applications that provide many categorized utilities, and where the user may benefit from sliding some of the content out of the way. Example 9-19 and Figure 9-11 demonstrate a neat layout combining Divided Boxes, relative sizing, and several style properties.

Example 9-19. *An advanced layout using Divided Boxes*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">
```

```

<s:layout>
  <s:HorizontalLayout horizontalAlign="center" verticalAlign="middle"/>
</s:layout>

<mx:HDividedBox width="70%" height="70%" backgroundColor="black"
  paddingTop="5" paddingBottom="5" paddingLeft="5" paddingRight="5">

  <s:BorderContainer height="100%" width="25%"
    borderColor="yellow" borderWidth="2" cornerRadius="4"
    backgroundAlpha="0.25">
    <!-- components go here -->
  </s:BorderContainer>

  <mx:VDividedBox width="75%">

    <s:BorderContainer height="100%" width="100%"
      borderColor="red" borderWidth="2" cornerRadius="4"
      backgroundAlpha="0.25">
      <!-- components go here -->
    </s:BorderContainer>

    <s:BorderContainer height="100%" width="100%"
      borderColor="green" borderWidth="2" cornerRadius="4"
      backgroundAlpha="0.25">
      <!-- components go here -->
    </s:BorderContainer>

    <s:BorderContainer height="100%" width="100%"
      borderColor="blue" borderWidth="2" cornerRadius="4"
      backgroundAlpha="0.25">
      <!-- components go here -->
    </s:BorderContainer>

  </mx:VDividedBox>

</mx:HDividedBox>

</s:Application>

```

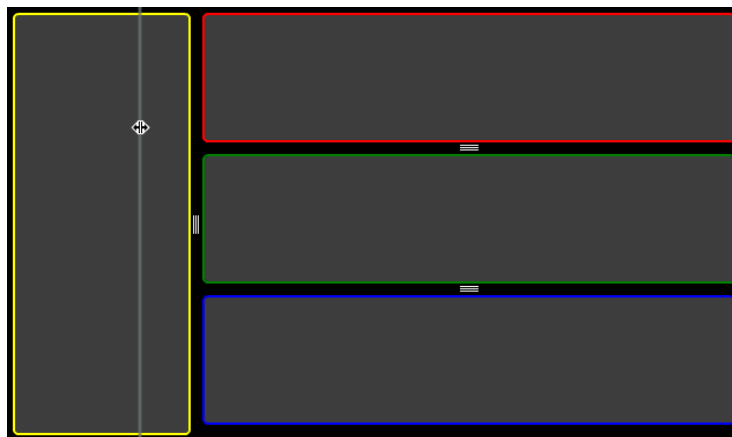


Figure 9-11. Resizing a content area in a Divided Box layout

TileLayout and TileGroup

If you want a layout that can really roll with the punches, the **TileLayout** is a great choice. Depending on your needs, you can either pair a nested **TileLayout** with a Spark container or use the ready-made **TileGroup** container.

The **TileLayout** arranges components dynamically. By default, it begins rendering any nested components horizontally until the available space is depleted, and then it starts a new row. This layout forces all tiles to have the same size, plus it will arrange them in a perfect grid. Unless you explicitly set a **tileWidth** or **tileHeight**, the largest child determines the size for all other tiles; consider this at work in Example 9-20 and Figure 9-12.

Example 9-20. Testing a **TileLayout** with asymmetrical components

```
<s:TileGroup height="100" width="400">
  <s:Label text="Antagonizing the TileGroup"/>
  <s:Button width="50" label="Wide=80"/>
  <s:Button width="150" label="Wide=150"/>
  <s:Button width="25%" label="Wide=25%"/>
  <s:Button height="80" width="80" label="High=80|Wide=80"/>
</s:TileGroup>
```

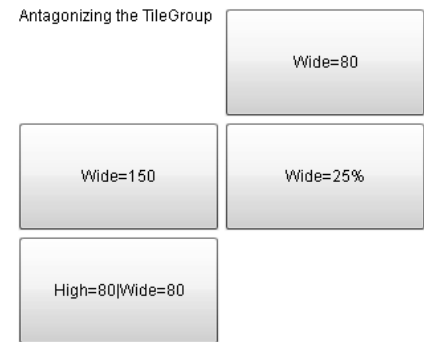


Figure 9-12. A **TileGroup** container imposing symmetry on several asymmetrical components

Form

The **Form** container creates a layout that resembles an HTML form, and it works in union with the **FormItem** wrapper-container; see Example 9-21 and Figure 9-13. **FormItem** wrappers take an input control such as a **TextInput**, provide that input control with a label, and create an aesthetic alignment of elements. Both the **Form** and the **FormItem** are limited to the Halo package.

Example 9-21. A **Form** container with **FormItem** wrappers providing labels for two **TextInput** controls

```
<mx:Form>
  <mx:FormItem label="Full Name:">
    <s:TextInput id="nameTI"/>
  </mx:FormItem>
  <mx:FormItem label="Email:">
    <s:TextInput id="emailTI"/>
  </mx:FormItem>
</mx:Form>
```

Full Name:

Email:

Figure 9-13. A simple **Form** layout with two input fields

A **Form** container with **FormItem** wrappers creates a layout like Figure 9-13, where the form fields stack vertically with right justification.

Form containers can also take a **FormHeading** control, which provides a title for the form. The **FormHeading** control will align with the form fields automatically.

NOTE

FormItems have additional functionality, such as their **required** property. We discuss this class in detail later in Chapter 10.

In Flash Builder's Design mode, forms are easy to create. Just drag a **Form** container into your **Application** container. Then, when you drag controls into the **Form**, such as **TextInputs**, **NumericSteppers**, or **DateFields**, Flash Builder automatically generates a **FormItem** wrapper, and you can double-click the **FormItem** to set its label. If you don't want a label at all, either leave it empty or go into Source mode and remove the **label** attribute from the **FormItem** altogether.

In order to align **Form** components properly, you must enclose them in a **FormItem** container. So, even if you don't want the label displayed, you still need the **FormItem** wrapper component.

Tabbing Through Fields

People generally expect to navigate an application using more than just the mouse. With that said, the most commonly used navigation alternative is the Tab key. Because so many people expect to use the Tab key to bounce through input fields, not accounting for this may lead to frustrated users.

Fortunately, Flex provides Tab navigation by default. Based upon a control's proximity to the currently selected field, the Tab key will shift focus to the next most reasonable field. The phrase *shift focus* is the programmer's term for "make active," and the **Form** container usually does a fine job of interpreting the right focus sequence.

However, complicated layouts with multiple forms occasionally confuse the native **FocusManager**, and you might require more fine-grained control. When this is the case, look for the **tabIndex** property. The **tabIndex** uses an integer value to define the order in which components should take focus through Tab navigation. For this to work properly, though, you must set the **tabIndex** of every control.

Spacers and Lines

There are a couple of additional components you occasionally might want to use in your layouts: the **Spacer** and the FXG **Line** graphic, which can help to occupy space and/or provide a visual separation between content areas.

Spacer

NOTE

*Sometimes it's advantageous to use the **Label** control like a **Spacer**. Because setting a **Label**'s **width** or **height** to a larger number doesn't affect text size, it can provide an additional means of controlling layout.*

The *Spacer* (`<mx:Spacer/>`) is an invisible control that, when used inside a layout container, will push other elements around for the purpose of holding space. The **Spacer** can take **width** and **height** assignments, as well as percentage-based sizes and minimum and maximum values.

A **Spacer** set inside an **HGroup** will separate its neighbor components. In Example 9-22, a **Spacer** that has a **width** of 100% creates a layout where the visual controls are pushed to opposite edges of the container, as demonstrated by Figure 9-14. It's worth mentioning that this arrangement is dependent on the fixed width of the **HGroup**.

Example 9-22. Absorbing space with the *Spacer* component

```
<s:HGroup width="250" height="100" verticalAlign="middle">
  <s:Button label="Left"/>
  <mx:Spacer width="100%" />
  <s:Button label="Right"/>
</s:HGroup>
```

**Figure 9-14.** A *Spacer* at work in an *HGroup*

Planning Your Layout

Flex's layout process performs a lot of calculation when determining component size and placement, so having too many nested containers can cause performance degradation. For that reason, it's worthwhile to streamline a layout to use only the containers you need.

Because the **Application** tag is a container itself, the following code, which aligns two **Buttons** vertically, isn't optimal:

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:VGroup>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
  </s:VGroup>

</s:Application>
```

You could reduce this code by removing the **VGroup** and declaring a **VerticalLayout** for your **Application**

container. By the way, this shows another format for declaring layouts:

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout><s:VerticalLayout/></s:layout>

  <s:Button label="Button 1"/>
  <s:Button label="Button 2"/>

</s:Application>
```

Like all aspects of software development, designing a great interface can benefit from a bit of planning. Creating a simple diagram of your application on paper can help, especially if you plan to use a lot of containers. The exercise will force you to think about how different components relate to one another. Again, the trick is to simplify and use only what you need. Doing so will create applications that run better as well as code that's easier to understand and maintain.

FXG Line Graphic

There might be times when you want to create a simple visual distinction between content areas of a layout. When this is the case, turn to the *FXG Line graphic*. We used **FXG Ellipse** and **Rect** graphics in Chapter 7 when we created the Collision! game, but this is our first look at the **Line**.

One aspect of the **Line** graphic that's counterintuitive is the need to wrap it inside a **Group** container for this sort of usage. Example 9-23 demonstrates use of the FXG **Line** as a content separator; see the result in Figure 9-15. Note that both the **Group** and the **Line** are set to 100% **width**, and the **Group** creates space on either side of the **Line** by having a fixed **height**. Furthermore, the **Line** is centered inside the **Group** using the **verticalCenter** constraint, which we've seen examples of throughout this book.

Example 9-23. *Applying an FXG Line within a Form*

```
<s:BorderContainer borderStyle="solid" borderColor="#000000">
  <mx:Form>
    <mx:FormHeading label="Contact Editor"/>
    <mx:FormItem label="First Name:">
      <s:TextInput/>
    </mx:FormItem>
    <mx:FormItem label="Last Name:">
      <s:TextInput/>
    </mx:FormItem>

    <s:Group width="100%" height="24">
      <s:Line width="100%" verticalCenter="0">
        <s:stroke>
          <s:SolidColorStroke color="black" weight="2"/>
        </s:stroke>
      </s:Line>
    </s:Group>

    <mx:FormItem label="Age">
      <s:NumericStepper minimum="0" maximum="125"/>
    </mx:FormItem>

    <s:Group width="100%" height="24">
      <s:Line width="100%" verticalCenter="0">
        <s:stroke>
          <s:SolidColorStroke color="black" weight="2"/>
        </s:stroke>
      </s:Line>
    </s:Group>

    <mx:FormItem label="Street:">
      <s:TextInput/>
    </mx:FormItem>
    <mx:FormItem label="Zip:">
      <s:TextInput/>
    </mx:FormItem>
  </mx:Form>
</s:BorderContainer>
```

Figure 9-15. FXG Line graphics separating sections of a Form container

Visualizing the Structure of Your Application

The Outline pane, which is accessible using either Design or Source mode, provides a great way to visualize and navigate the hierarchy of your application. Shown in Figure 9-16, and accessible by selecting Window→Outline (if not already visible in your workspace), you can see the structure of your application in a tree list. When you select an item in this list, the selection is matched in the Editor, and vice versa.

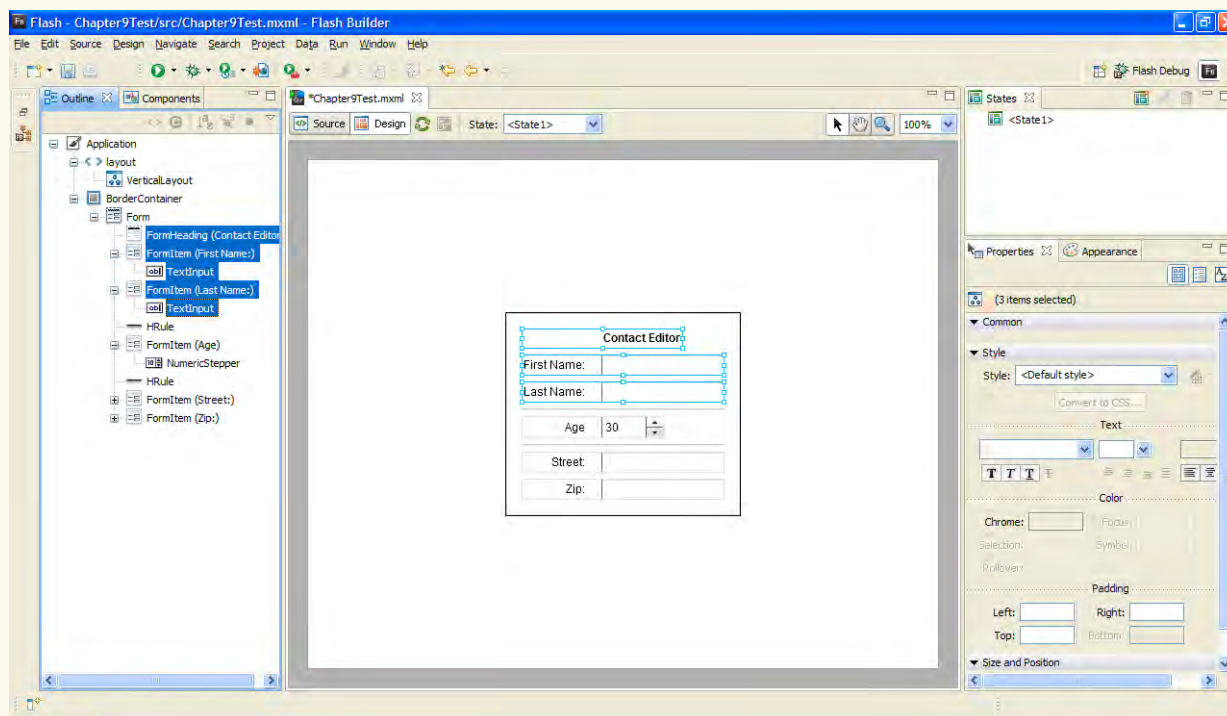


Figure 9-16. Selecting components in the Outline pane selects components in the Editor

Alignment

Both the **HorizontalLayout** and **VerticalLayout**, as well as the **HGroup** and **VGroup** containers that implement them, support horizontal and vertical alignment styling. You set the style type using either the **horizontalAlign** or **verticalAlign** properties, respectively.

Consider a **BorderContainer** having a **width** and **height** of 200 pixels, using a **VerticalLayout**, and containing three buttons. By default, the **verticalAlign**

is set to **top** and the **horizontalAlign** is set to **left**, meaning the buttons will be aligned along the top and left edges of the container. You can make them centered, though, by changing the **verticalAlign** to **middle** and the **horizontalAlign** to **center**, as shown in Example 9-24 and Figure 9-17.

Example 9-24. *Centering components using the **verticalAlign** and **horizontalAlign** properties of the **VerticalLayout***

```
<s:BorderContainer width="200" height="200" borderColor="#000000">
  <s:layout>
    <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
  </s:layout>
  <mx:Button label="Button"/>
  <mx:Button label="Button"/>
  <mx:Button label="Button"/>
</s:BorderContainer>
```

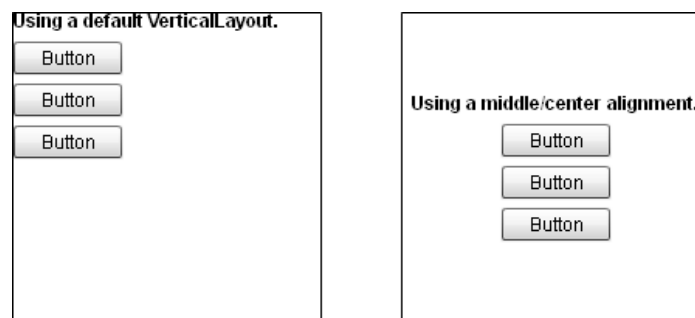


Figure 9-17. *The **VerticalLayout** with default alignment, and as middle/center alignment*

WARNING

Alignment typically doesn't make a difference when no explicit size has been set on the container. A container with no size will simply fit its contents, and so alignment might not be apparent, because there would be no space between the container and its children.

Supposing that you won't always want to use a centered alignment, the following values are available to the **horizontalAlign** and **verticalAlign** properties:

horizontalAlign

The possible values include: **center**, **content**, **justify**, **justify**, **left**, and **right**.

verticalAlign

The possible values include: **bottom**, **middle**, and **top**.

Constraints-Based Layout

Another powerful approach to building your application, and one that works well when placing nested layout containers, is the constraints-based layout.

Constraints Explained

Think of constraints as an extension of the **BasicLayout**. This makes sense because a component's parent must be set up for absolute positioning in order for constraints to work. Essentially, *constraints* anchor a component a set distance, in pixels, from a corresponding position within its parent container. The effect of this layout tactic is most pronounced by resizing, which causes the constrained component to move relative to its anchor points.

top, bottom, left, and right

The first four constraint properties correspond to the four edges of a rectangle: **top**, **bottom**, **left**, and **right**. For a quick example of how they're used, look back at the color swatches code from earlier in the chapter; both the **Button** and the **CheckBox** were positioned using constraints.

There's nothing mystical or deceptive about the constraint properties. It should be fairly obvious that **right="5"** and **bottom="5"** means "five pixels from the right, and five pixels from the bottom."

To get a feel for the practical advantage offered by constraints, build the simple application in Example 9-25 and have some fun resizing your browser window (Figure 9-18).

Example 9-25. *Using constraints to anchor components into the corners*

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:BorderContainer top="5" right="5"
    borderStyle="solid" borderWidth="5" borderColor="#FF0000"/>

  <s:BorderContainer bottom="5" right="5"
    borderStyle="solid" borderWidth="5" borderColor="#00FF00"/>

  <s:BorderContainer bottom="5" left="5"
    borderStyle="solid" borderWidth="5" borderColor="#FFFF00"/>

  <s:BorderContainer top="5" left="5"
    borderStyle="solid" borderWidth="5" borderColor="#0000FF"/>

</s:Application>
```

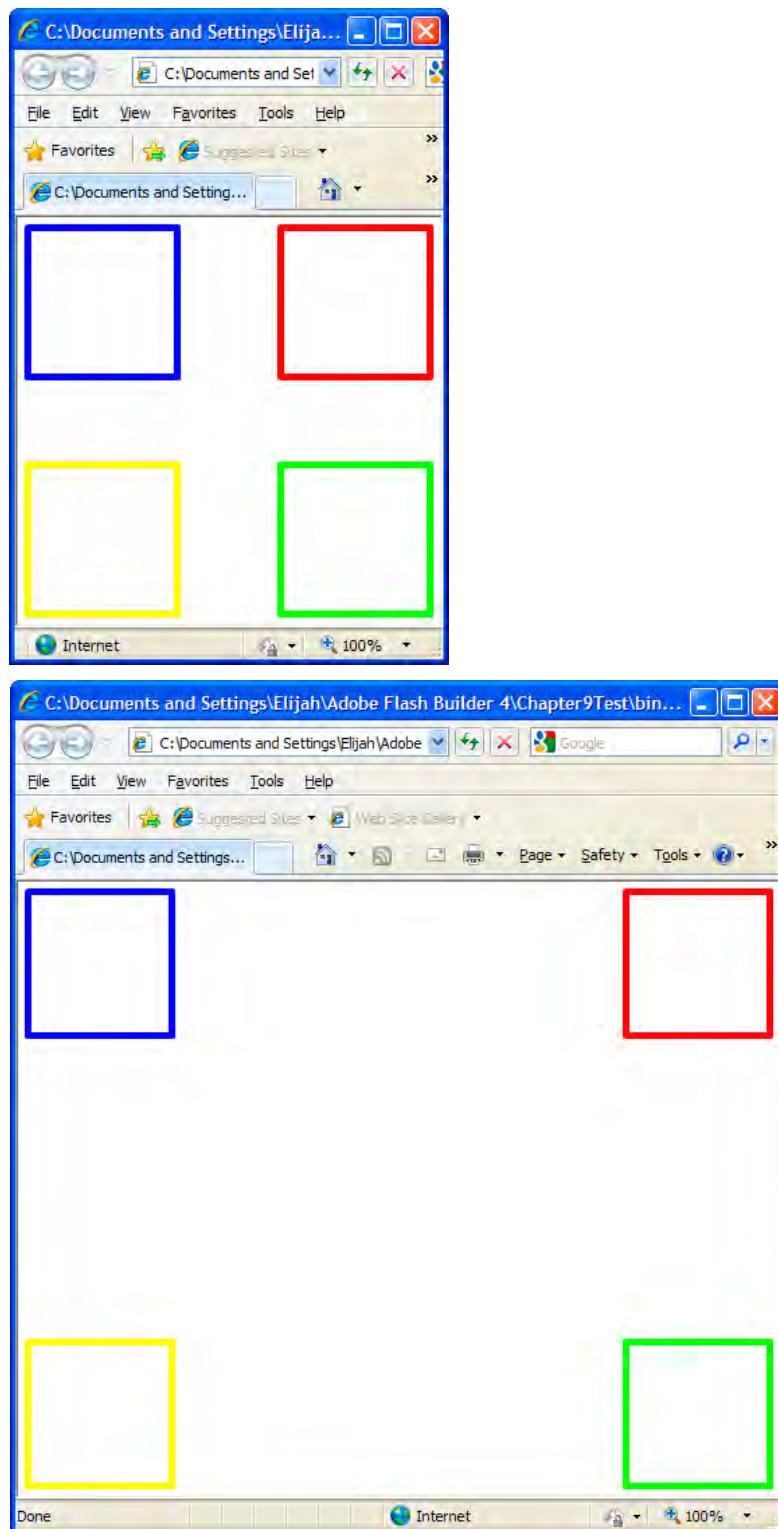


Figure 9-18. Anchoring components to the corners using constraints

horizontalCenter and verticalCenter

Two more constraints, which we've used all along, are the centralizing properties: **horizontalCenter** and **verticalCenter**. Both of these properties take positive or negative integers and use them to position a component relative to its parent container's center.

If a container is using Absolute/Basic layout, the properties **horizontal="0"** and **vertical="0"** will position a component in the middle of its parent. If you assign these properties positive integers—let's suppose **horizontal="15"** **vertical="15"**—the renderer will place your component 15 pixels left of center and 15 pixels above center. By contrast, if you used negative integers—**horizontal="-15"** **vertical="-15"** in this instance—the renderer would shift your component 15 pixels right of center and 15 pixels below center.

We've used these properties enough in prior examples that you should be able to apply them effectively without further demonstration.

NOTE

Mixing **left** or **right** anchors with **horizontalCenter** can lead to unpredictable results. Similarly, **top** and **bottom** anchors don't mix well with **verticalCenter**. Although they're not mutually exclusive, it just doesn't make sense to anchor a component to both the left edge and the center of its parent. But, if you do specify both a center constraint and an edge constraint, the size of the component will be calculated from the edge constraint, whereas its position will be determined by the center constraint.

Assigning Constraints in Design Mode

It's easy to apply constraint settings in Design mode. Assuming a container is using a Basic/Absolute layout, select any component you add to it, and with the Properties pane in Standard View, scroll down to the Size and Position settings. Under the Constraints heading, check the constraint properties you want to apply, then establish their values. In Figure 9-19, we're using Design mode to set bottom and left constraints for the green BorderContainer component in Example 9-25.

Do note that you will not see Constraints options for components whose parent containers do not implement the Basic or Absolute layouts.

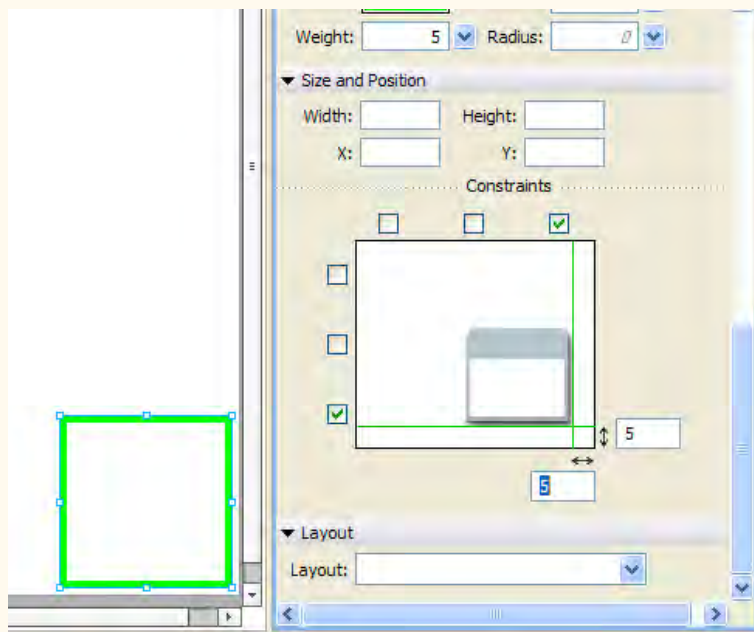


Figure 9-19. Applying constraints in Design mode

Bindings in Layout

In Chapter 8, we briefly mentioned that you can use bindings to create dynamic layouts. Just as you can bind two **text** properties together, you can also bind two containers' positioning and/or size properties together, ensuring they occupy space relative to one another.

The following example presents two **Panel** containers. The first is positioned and sized casually enough; however, the **x**-value of the second **Panel** is bound to the **x** + **width** + 10 of the first **Panel**:

```
<s:Panel id="onePanel" x="100" y="100"
        width="200" height="200"/>

<s:Panel id="anotherPanel"
        x="{onePanel.x + onePanel.width + 10}"
        y="100" width="200" height="200"/>
```

One caveat of this approach is that, due to the bindings, Flash Builder won't be able to show the layout in Design mode.

Summary

In this chapter, you learned a great deal about arranging layouts in Flex. You dove into the general layout types and learned how to manage component content with the Display List and ActionScript. You reviewed the concepts of explicit and percentage-based sizing, and you gained a deeper respect for the work Flex handles when rendering your layouts. You learned about a few new containers, and you also learned some tactics for constructing layouts using constraints and bindings.

There was plenty to cover in this chapter, but everything you've learned has an immediate use. In the next chapter, you'll learn about the **Form** container and how you can automate input formatting and validation on the fly.

CREATING RICH FORMS

“Fast is fine, but accuracy is everything.”

—Wyatt Earp

Have you ever filled out an HTML form on a website, submitted it, and waited for the result—only to find that one of the fields had an error? How easy was it to find your mistake? Was it something silly like not putting parentheses around the area code of a phone number, or perhaps adding them when they weren’t needed? Wouldn’t it be nice if that never happened again?

With Flex, validating and formatting user input is a cinch. Built into the most common controls are helpful methods that provide feedback when user-submitted values are problematic; similarly, it’s easy to link Flex inputs with nonvisual formatter components that automatically sculpt data into preferred formats. Working together, Flex validators and formatters give your applications a helpful, responsive UI. As a bonus, of course, cleaner user input also means a cleaner database.

In this chapter, we create an input form that collects contact information for the purpose of demonstrating data validation and input formatting techniques.

Preparing a Form-Based Application

To get started, we construct a form layout complete with first and last name, email, phone, address, and zip code fields, and for the fun of it, we also play with the **DateField** and **ColorPicker** controls.

As usual, create a new project in your workspace; this time, name it *RichForms*.

You have the option of constructing the form layout in either Design or Source mode. Most of the UI can be established in Design mode, and that’s how we describe it in the following section. However, if you prefer to develop in Source mode, you can compare your code to Example 10-3, which you’ll find in the section “The finished form layout” on page 183.

IN THIS CHAPTER

Preparing a Form-Based Application

Validating Data

Restricting Input

Formatting Input

Combining Restrictions and Formatters

Linking Formatters to Functions

Summary

Starting the Form

Assuming you're in Design mode with the Properties pane in Standard view, start by deleting the application's default **height** and **width** values, and then set the application layout to **spark.layouts.VerticalLayout**. Next, switch the Properties pane to Category view, expand the Layout options, and set **horizontalAlign** equal to **center** and **verticalAlign** equal to **middle**. That handles the application container.

NOTE

*When you use Design mode to drop controls into a **Form**, Flash Builder automatically wraps the controls in **FormItem** containers.*

Now drag a **Form** container from the Components pane into your application container. Once you drop the **Form** into your application, Flash Builder will ask you to assign the form's dimensions. Set the **width** value to 324, but leave the **height** as **<fit to content>**. The default **height** value ensures your form will grow as you add components to it. Finally, in the Properties pane, set the form's **id** to **contactForm**.

Next, drag a **FormHeading** into your form. You'll find it in the Components pane, just below the **Form** container. After you drop it, give it the **Label** "Contact Details" or some other suitable moniker.

Now drag four **TextInput** controls to your **Form**. At this point you should have something resembling Figure 10-1.

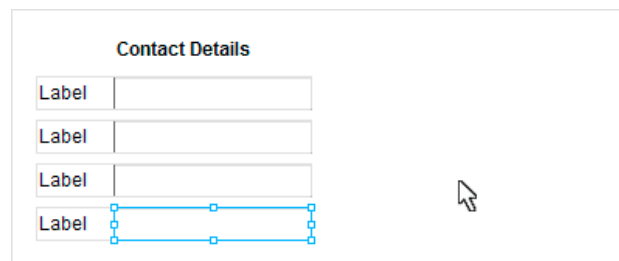


Figure 10-1. Four **TextInput** controls hanging out in the new **Form**

For the sake of making a point, we want you to jump into Source mode and review the code generated by Flash Builder (we formatted Example 10-1 a little to help readability).

Example 10-1. The initial form code generated by Design mode

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
  </s:layout>
```



```

<fx:Declarations>
    <!-- non-visual elements (e.g., services, value objects) -->
</fx:Declarations>

<mx:Form width="324" id="contactForm">
    <mx:FormHeading label="Contact Details"/>
    <mx:FormItem label="Label">
        <s:TextInput/>
    </mx:FormItem>
    <mx:FormItem label="Label">
        <s:TextInput/>
    </mx:FormItem>
    <mx:FormItem label="Label">
        <s:TextInput/>
    </mx:FormItem>
    <mx:FormItem label="Label">
        <s:TextInput/>
    </mx:FormItem>
</mx:Form>

</s:Application>

```

This code gives us a couple of points to discuss. First, we're using several components from the Halo namespace—**Form**, **FormHeading**, and **FormItem**—because these components do not yet have Spark counterparts. Second, although we dragged four **TextInput** controls into our form, Flash Builder gave us four nicely wrapped **FormItem** components. Notice that the **TextInput** controls we added became children of the **FormItem** components.

This second point is important because adding **id** and **label** values in Design mode requires selecting the proper component. Originally you added four **TextInput** controls, but now you have eight total components. So, as you assign component properties, make sure the correct component is selected. Adding an **id** of **firstNameTI** to the **id** property of a **FormItem** won't achieve the desired results.

While you're in Source mode, establish the following four labels for these **FormItem** controls: First Name, Last Name, Email, and Phone. While you're at it, give each **TextInput** control an **id** value. We used **firstNameTI**, **lastNameTI**, **emailTI**, and **phoneTI**.

If you prefer to use Design mode, use the Outline pane to choose between the **FormItem** components and the **TextInput** controls while you're setting their respective property values.

Making Inputs Required

Let's return to Design mode, where we'll introduce a new property—**required**.

Starting with the First Name **FormItem** container, select it using the Outline pane, and in the Properties pane, find the **required** property and set it to **true**. As you can see in Figure 10-2, the **required** property adds a red asterisk (*) between the **FormItem** container's label and its child component, informing anyone using your application of the importance of that field.

NOTE

*When working with forms, where controls and **FormItem** components are tightly packed, you may find it more convenient to establish the various **label** and **id** values in Source mode. If you want to remain in Design mode, though, make sure to use the Outline pane to select components.*

Repeat this step for the Email **FormItem**, making it required as well.

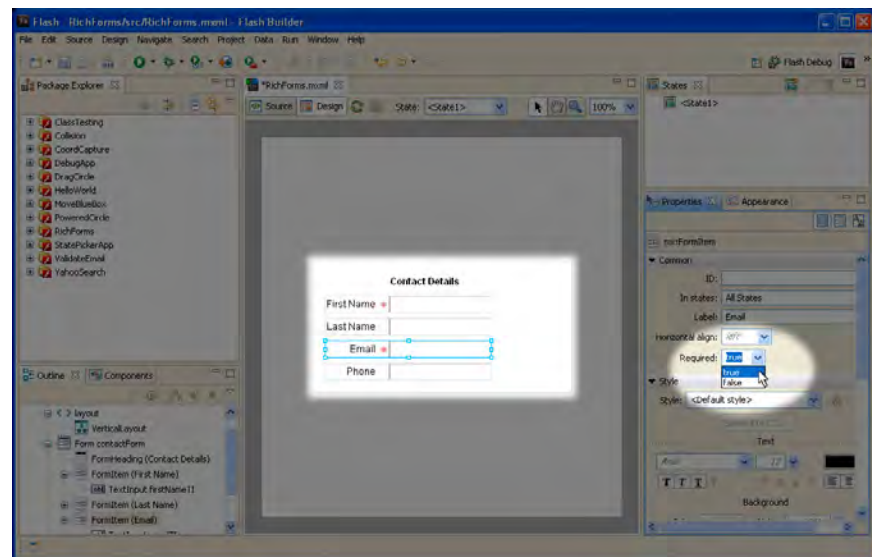


Figure 10-2. Setting the required property on a selected **FormItem** container

Although the **required** property denotes important form fields, it does not interactively enforce them; the red asterisk only signals importance. Fortunately, you can use Flex validators to ensure proper input, and we discuss the validator components later in this chapter.

Adding a **RadioButtonGroup**

We need a solution to select the type of phone number being entered, with choices including “mobile”, “home”, and “other”. Since we have three values, a simple **CheckBox** will not work, as it provides only two alternatives: on (**true**) or off (**false**). On the other hand, radio buttons are great for letting someone select one of a few values, so we’ll create a set of radio buttons to handle the phone type.

Assuming you’re still in Design mode, drag a **RadioButtonGroup** into the Phone **FormItem**. You’ll be prompted with a dialog box similar to Figure 10-3, which helps configure the group. Give the group a name of **phoneRadioButtonGroup**, and remove the default buttons (i.e., Button1, Button2) by selecting them and clicking on Remove. With a clean slate, use the Add button to create three new radio buttons corresponding to the three phone types: mobile, home, and other. When you’re finished, click OK, and Flash Builder will generate your radio buttons.

While still in Design mode, select the first radio button (probably “mobile”) and set its **selected** property to **true**. Now your application will assume any

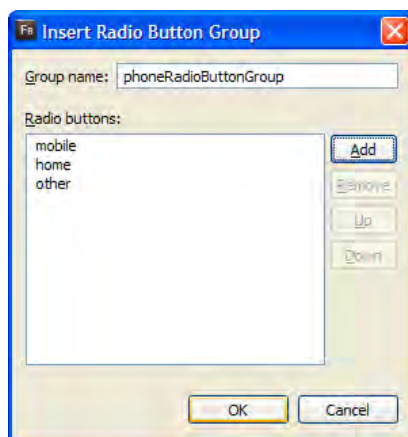


Figure 10-3. The Insert Radio Button Group dialog box

phone number provided is a mobile number, unless the user toggles a different radio button.

Let's jump into Source mode to see how this affects the code, which is shown in Example 10-2. Notice that an `<s:RadioButtonGroup/>` tag was added in the Declarations section and three `<s:RadioButton/>` tags were added within the Phone `FormItem`.

Example 10-2. The code after adding a `RadioButtonGroup` in Design mode

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
  </s:layout>

  <fx:Declarations>
    <s:RadioButtonGroup id="phoneRadioButtonGroup"/>
  </fx:Declarations>

  <mx:Form width="324" id="contactForm">
    <mx:FormHeading label="Contact Details"/>
    <mx:FormItem label="First Name" required="true">
      <s:TextInput id="firstNameTI"/>
    </mx:FormItem>
    <mx:FormItem label="Last Name">
      <s:TextInput id="lastNameTI"/>
    </mx:FormItem>
    <mx:FormItem label="Email" required="true">
      <s:TextInput id="emailTI"/>
    </mx:FormItem>
    <mx:FormItem label="Phone">
      <s:TextInput id="phoneTI"/>
      <s:RadioButton label="mobile" groupName="phoneRadioButtonGroup"
        selected="true"/>
      <s:RadioButton label="home" groupName="phoneRadioButtonGroup"/>
      <s:RadioButton label="other" groupName="phoneRadioButtonGroup"/>
    </mx:FormItem>
  </mx:Form>

</s:Application>
```

The `<s:RadioButtonGroup/>` tag isn't actually a container for the `RadioButton` controls; rather, it's a *nonvisual* component that unites several `RadioButton` objects, allowing you to reference them as a group. It also allows the buttons in the group to function together; for example, only one radio button can be selected at any given time. Being a nonvisual component, the `RadioButtonGroup` belongs in the **Declarations** section. Radio buttons link to a particular group through their `groupName` property, and in this case, all of our radio buttons belong to `phoneRadioButtonGroup`.

Completing the Form Layout

To finish our **Form** layout, we'll add controls for Address, ZIP Code, Birthday, and Favorite Color.

Address and ZIP Code

For the Address, drag a **TextArea** control out of the Components pane and drop it below the Phone **FormItem**. Using the Outline pane, select the new **FormItem** and give it the **label** "Address"; next, give the **TextArea** an **id** of **addressTA**. Since the default **TextArea** is taller than we require, set its **height** to 75.

Is the lack of **width** symmetry between the form controls vexing you? If so, you could set the **width** values for the **TextInput** controls to 188, which is the default **width** for the **TextArea**. On the other hand, you could set all the various text controls, including the **TextArea**, to another arbitrary value. For this example, we set our **TextInput** controls to a **width** of 188.

For the zip code, drag another **TextInput** into your form and drop it below the Address **FormItem**. Provide the **label** "ZIP Code" to the new **FormItem** and give the **TextInput** an **id** of **zipCodeTI**. If necessary, set the **width** of the new **TextInput**.

DateField

The Birthday input will use a **DateField** control, which allows for easy selection of calendar dates. When someone clicks on the **DateField** control, a mini-calendar will pop out, allowing the user to choose a specific date.

We don't need special code to add the **DateField**, so simply drag it out of the Components pane and drop it below the ZIP Code **FormItem**. Label the new **FormItem** "Birthday" and assign the control an **id** of **dateField**. To maintain visual symmetry, you may want to set the **width** of the **DateField**.

With the **DateField** control still selected, switch the Properties pane to Category view, expand Other, and scroll to the bottom of the list until you find the property **yearNavigationEnabled**. Assign it a value of **true**. This property makes it easy to advance the calendar one year at a time, which will certainly be necessary since people will be selecting dates that are a few years back.

For additional discussion of the **DateField** control, see the sidebar titled "More About the DateField" on page 184.

ColorPicker

We know that favorite colors have nothing to do with contact information. However, we wanted to include the **ColorPicker** somewhere, so we decided to add it in this context. Like the **DateField**, the **ColorPicker** is a self-contained, no-nonsense component. Just drag it into your form. Give its **FormItem** the label “Favorite Color” and give the **ColorPicker** itself an **id** of **colorPicker**. Once again, just for the sake of symmetrical controls, we gave the **ColorPicker** a **width** of 188.

The finished form layout

At this point, we’re finished designing the form’s layout. If you jump into Source mode, your code should resemble Example 10-3.

Example 10-3. *The source code for the form layout*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
  </s:layout>

  <fx:Declarations>
    <s:RadioButtonGroup id="phoneRadioButtonGroup"/>
  </fx:Declarations>

  <mx:Form width="324" id="contactForm">
    <mx:FormHeading label="Contact Details"/>
    <mx:FormItem label="First Name" required="true">
      <s:TextInput id="firstNameTI" width="188"/>
    </mx:FormItem>
    <mx:FormItem label="Last Name">
      <s:TextInput id="lastNameTI" width="188"/>
    </mx:FormItem>
    <mx:FormItem label="Email" required="true">
      <s:TextInput id="emailTI" width="188"/>
    </mx:FormItem>
    <mx:FormItem label="Phone">
      <s:TextInput id="phoneTI" width="188"/>
      <s:RadioButton label="mobile" groupName="phoneRadioButtonGroup"
        selected="true"/>
      <s:RadioButton label="home" groupName="phoneRadioButtonGroup"/>
      <s:RadioButton label="other" groupName="phoneRadioButtongroup"/>
    </mx:FormItem>
    <mx:FormItem label="Address">
      <s:TextArea height="75"/>
    </mx:FormItem>
    <mx:FormItem label="ZIP Code">
      <s:TextInput id="zipCodeTI" width="188"/>
    </mx:FormItem>
  </mx:Form>
</s:Application>
```

Contact Details

First Name *

Last Name


Email *

Phone

☒ mobile
☐ home
☐ other

Address

ZIP Code

Birthday 

Favorite Color

```
<mx:FormItem label="Birthday">
  <mx:DateField id="dateField" width="188"
    yearNavigationEnabled="true"/>
</mx:FormItem>
<mx:FormItem label="Favorite Color">
  <mx:ColorPicker width="188" id="colorPicker"/>
</mx:FormItem>
</mx:Form>
```

```
</s:Application>
```

If you run the application, it should resemble Figure 10-4.

Validating Data

This section demonstrates how the Flex validator components can not only ensure you're getting the data you require, but also inform users when their data input has problems that require fixing before they can proceed.

Because validators are nonvisual components, you need to be in Source mode in order to add validators to an application.

Figure 10-4. The emerging Contact Details form

More About the DateField

You might appreciate knowing about additional properties available to the **DateField** control.

```
MinYear="1992"
```

By default, the **DateField** allows calendar selection between the years 1900 and 2100, and although this will suit most applications, some scenarios may require imposing a reduced or expanded range of years.

```
MaxYear="2011"
```

Like the **MinYear**, the **MaxYear** field allows you to set a cap on the maximum year exposed by the calendar.

```
yearNavigationEnabled="true"
```

This attribute allows users to rapidly advance year values in the mini-calendar. With this property disabled, all 12 months would need to be advanced in order to change the year.

```
editable="true"
```

The **editable** property allows users of your application to manually enter dates into the **DateField** control. There is a hidden benefit to using an editable **DateField**. Specifically, the **DateField** has built-in data validation and parsing, which allows it to accept many different date formats. Test this feature by setting the **editable** property of your **DateField** to **true**, and then run your application and throw a few dates at it.

Try entering your own birthday in the following formats: MM/DD/YY, MM-DD-YY, MM DD YY, MM/DD/YYYY, etc. As long as you enter the month before the date, it just works. On the other hand, the editable **DateField** does *not* accept month names or abbreviations.

```
formatString="DD-MM-YYYY"
```

The **formatString** property lets you assign a **String** value to display the date in a preferred format. For instance, if you prefer to see dates as day, month, then year, you could use the format in the example property assignment shown here. One caveat, though: while the date will be formatted differently for display, use of a **formatString** will cause the **DateField** to require manual date entry (i.e., **editable="true"**) in the same format.

```
selectedDate
```

Although the **DateField** looks as though it contains basic text, it actually uses a **selectedDate** property that is in fact a **Date** object. **Date** is a complex data type that is used just for dates and times. It represents a single moment down to the millisecond. When you type in a date that is validated by the **DateField** control, or when you select a date using the mini-calendar, the **selectedDate** property is filled with a **Date** object.

Using Validators

Flex *validator* components can handle the following evaluations right in the UI:

- Check whether a source control has received input
- Test user input for minimum/maximum length requirements
- Provide custom error messages on controls having invalid input
- Provide built-in error messages on controls receiving common data types, such as email and zip code

To learn data validation hands-on, we'll go down the line, establishing Flex validator components for each of the fields in the **RichForms** application. You'll start with the First Name field, which we designated as **required** via its **FormItem**. Because the **FormItem** doesn't handle error checking, we'll create a **StringValidator** to handle the validation functionality.

StringValidator

A **StringValidator** is a basic validator that ensures text has been entered. To use it, create a `<mx:StringValidator/>` tag in the **Declarations** section of your MXML file. The `<mx:StringValidator/>` tag has a **source** attribute that identifies the control you want it to watch. The **property** attribute tells it which **property** of a control should be validated. A **StringValidator** watching **firstNameTI** in the **RichForms** application is emphasized in Example 10-4.

Example 10-4. Adding a *StringValidator* to the *Declarations* section

```
<fx:Declarations>
    <s:RadioButtonGroup id="phoneRadioButtonGroup"/>
    <mx:StringValidator id="firstNameValidator"
        source="{firstNameTI}" property="text"/>
</fx:Declarations>
```

Now run the application and see how validation operates. If you give focus to the First Name field, then fail to enter a value before tabbing to or clicking on another input, a red border will surround **firstNameTI**, signaling something is amiss. Then, if your mouse hovers over that field, a message will appear explaining what is wrong. By default the **StringValidator** warns, "This field is required."

But you can do better than that. Using the validator's **requiredFieldError** property, you can enter a custom message. For a more descriptive message, add a **requiredFieldError** attribute like the one in Example 10-5.

NOTE

*As nonvisual components, your validator objects should be added within the **Declarations** section of your code. This is true for all of the validator classes.*

WARNING

The **source** property of a validator component is bound directly to a control. This is unusual compared to other bindings we've seen. In comparison with the correct implementation in Example 10-4, the following code would be incorrect:

```
<mx:StringValidator
    id="firstNameValidator"
    source="{firstNameTI.text}"/>
```

NOTE

When you click on a control or when you use the Tab key to move to the next control, that control is said to have gained focus or become activated. A control should have focus before it can receive keyboard input.

Example 10-5. Adding a `requiredFieldError` message to the `StringValidator`

```
<mx:StringValidator id="firstNameValidator"
  source="{firstNameTI}" property="text"
  requiredFieldError="What we have here..&#13;is a failure to communicate."/>
```

Just to clarify, `` is an XML character reference, which is an approach for inserting special characters, or in this case, a line break.

Now run the application to test the `StringValidator`; you should see a form like the one shown in Figure 10-5.

Contact Details

First Name *

Last Name

Email *

Phone

☒ mobile
☐ home
☐ other

Address

ZIP Code

Birthday

Favorite Color

Figure 10-5. The user-friendly form

We'll continue by validating the Last Name field using slightly more specific conditions. Although it isn't necessary to provide a last name for this form, if one is entered, we want to validate it. We'll handle this with another `StringValidator`, this time setting its `required` property to `false` (it defaults to `true`) and its `minLength` property to 2. This arrangement will fire only if someone supplies a Last Name value, and if he does, the validator will ensure the input is at least two characters long. This could prevent someone from inputting a last initial when you want his complete last name.

As with Example 10-5, you can customize the error message displayed by this validator. You won't customize it by using a `requiredFieldError`, however, because we want this message to display when the text entered is too short. To accomplish this, use the `tooShortError` property. Example 10-6 shows the validator we created for the Last Name field.

Example 10-6. Adding a `tooShortError` message to a `StringValidator`

```
<mx:StringValidator id="lastNameValidator"
  source="{lastNameTI}" property="text"
  required="false" minLength="2"
  tooShortError="What kind of last name is that?"/>
```

Now run the application. Your results should resemble Figure 10-6.

Figure 10-6. The increasingly demanding application

EmailValidator

Our next item is the Email field. To validate an email address, we'll use the handy **EmailValidator**. It works like the **StringValidator**; all you need to do is bind it to the `emailTI` control. You can customize the `requiredFieldError` message if you want, but the **EmailValidator** filters several additional errors. For example, it has a `missingAtSignError` message, an `invalidDomainError`, and everything in between. Because the **EmailValidator** provides a thorough complement of default error messages, and since the standard messages are appropriate, we won't customize them. But it's nice to know customized messages are available to you. Example 10-7 provides the code to add an **EmailValidator** to the RichForms application.

Example 10-7. Adding an `EmailValidator` to the RichForms application

```
<mx:EmailValidator id="emailValidator"
  source="{emailTI}" property="text"/>
```

Once again, run the application. It should resemble Figure 10-7.

WARNING

There could be instances where a user's last name is actually one letter long, so the validation in Example 10-6 would create a problem for these users. When you're designing validation schemes, always try to think of every possibility you might encounter from your users.

NOTE

EmailValidator and **PhoneValidator** cannot verify whether an email address or a phone number truly exist in the real world. Like all validators, they simply ensure input is arranged using a properly formatted character pattern.

Contact Details

First Name * Stazmo

Last Name Abernathy


Email * staz An at sign (@) is missing in your e-mail address.

Phone

☒ mobile
☐ home
☐ other

Address

ZIP Code

Birthday 

Favorite Color

Figure 10-7. A precise error message from the `EmailValidator`

PhoneNumberValidator

Now we'll validate the Phone Number field. If you haven't already guessed, we'll use a `PhoneNumberValidator` this time, shown in Example 10-8. This is another case where the `FormItem` does not designate the field as **required**, but you should validate the input anyway.

Example 10-8. The code for a `PhoneNumberValidator`

```
<mx:PhoneNumberValidator id="phoneValidator"
    source="{phoneTI}" property="text"
    required="false"/>
```

By default, the `PhoneValidator` will accept many phone number formats without complaining. That means (415)555-8273, 415-5558273, or even 415 555 8273 are all acceptable inputs; however, entering "I don't have one" won't pass, as shown in Figure 10-8.

ZipCodeValidator

You don't need to validate the Address field, but you could validate it using the same approach we applied to the Last Name field, ensuring any input supplied meets a specified length. Either way, our next example will validate the zip code, and in keeping with our current pattern, we'll use a `ZipCodeValidator` for this task.

Contact Details

First Name * Stazmo

Last Name Abernathy

Email * staz@acmecodeworks.com

Phone I don't have one

☒ mobile
☐ home
☐ other

Address

ZIP Code

Birthday

Favorite Color

Your telephone number contains invalid characters.

NOTE

For many client-server applications, validating just on the client side isn't enough. Generally, these applications will have a server-side validation scheme as well, ensuring that no improper information is stored on the server.

Figure 10-8. No getting past the PhoneValidator

The **ZipCodeValidator** can validate either U.S. zip codes (the default) or both U.S. and Canadian codes. You specify this using the **domain** property, which takes a string value of either “US Only” or “US or Canada”. Our example demonstrates the latter, handling both U.S. zip codes and Canadian postal codes. This is one of those cases where we'll handle the property assignment using a *constant*. You learned about constants in Chapter 7, but if you need a refresher, return to that chapter and review the section titled “Event Constants” on page 114.

We'll assign the validator's **domain** property using a constant that's available on the **mx.validators.ZipCodeValidatorDomainType** class. Within a binding, supply the constant **ZipCodeValidatorDomainType.US_OR_CANADA** for the value of the **domain** property. Using code completion in Flash Builder, it should be easy to assign the **domain** property and automatically import the **ZipCodeValidatorDomainType** class. However, if code completion gives you a fit, you can always create the **import** statement manually at the top of the Script/CDATA block. Example 10-9 shows both the **import** statement and the **ZipCodeValidator** declaration.

Example 10-9. The Script and Declarations code necessary for the **ZipCodeValidator** component

```
<fx:Script>
    <![CDATA[
        import mx.validators.ZipCodeValidatorDomainType;
    ]]>
</fx:Script>
```

```
<fx:Declarations>

    <!-- other validators and declarations above this comment -->

    <mx:ZipCodeValidator id="zipCodeValidator"
        source="{zipCodeTI}" property="text" required="false"
        domain="{ZipCodeValidatorDomainType.US_OR_CANADA}"/>

</fx:Declarations>
```

Now the form will validate U.S. zip codes and Canadian postal codes, as shown in Figure 10-9.

Contact Details

First Name * Stazmo

Last Name Abernathy

Email * staz@acmecodeworks.com

Phone 4155558273

☒ mobile
☐ home
☐ other

Address

ZIP Code ABCDEFG| The Canadian postal code must be formatted 'A1B 2C3'.

Birthday

Favorite Color

Figure 10-9. The `ZipCodeValidator` even speaks Canadian

NOTE

There is even a `RegExpValidator` you can use to build very powerful validation routines using regular expressions, which are used to analyze special patterns in text.

Additional validator components

You don't need to validate the `DateField`, because that control handles itself. However, there is a `DateValidator` component that's available to you if you need it for other purposes. In addition to the `DateValidator` and the validator components we've demonstrated in this section, Flex provides several other validators you can take advantage of, such as the `CreditCardValidator`, `CurrencyValidator`, `NumberValidator`, and `SocialSecurityValidator`.

Creating That Error Look Using the `errorString` Property

You've discovered how to use validators to display helpful error messages on the Flex UI controls, but there's another way to display an error message on a control without using a special validator. With the **`errorString`** property, you can force the red border and pop-up message to display. This approach is particularly useful when combined with `ActionScript` to examine special criteria.

For example, you could give the **`ColorPicker`** an error message using the following code combination:

```
<fx:Script>
    <![CDATA[
        private function onColorSelection():void{
            if(colorPicker.selectedColor == 10027263){
                colorPicker.errorString = "Hmm.. " +
                    "not the color we expected.";
            }else{
                colorPicker.errorString = "";
            }
        }
    ]]>
</fx:Script>

<mx:ColorPicker width="188" id="colorPicker"
    change="onColorSelection()"/>
```

This example takes the **`uint`** value returned by the **`selectedColor`** property of the **`ColorPicker`** and, if the value matches the shade of purple being selected on the left of Figure 10-10, the script creates an error prompt like the one you see on the right side of the figure. In the script, if the error condition isn't met, the **`errorString`** property is reset to an empty string (`""`), which removes the error indicators.

By the way, we determined the **`uint`** value returned by the **`ColorPicker`** for that color selection by tracing the value to the console, which we discussed in Chapter 6.

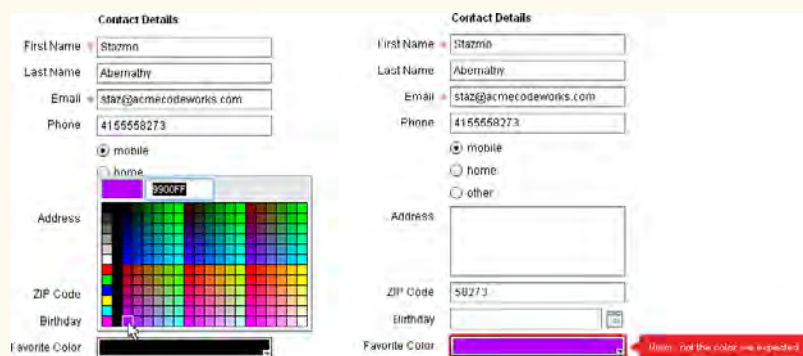


Figure 10-10. Using script and the `errorString` property to create a truly unique error prompt

Custom Validation Techniques

Have you noticed that validation components perform their validation when focus is moved from one input control to the next? This section discusses three additional mechanisms you can use to control validators, specifically their **triggerEvent** and **trigger** properties and their **validate** method.

The **triggerEvent** and **trigger** properties

Each validator component has a **triggerEvent** and a **trigger** property that can override focus change as the event controlling validation. The **triggerEvent** links the validator to a specific event fired by its **source** control. On the other hand, the **trigger** property is used to override the **source** control if a specific UI component should trigger validation. Let's consider some examples.

To cause the **EmailValidator** to test email input in real time, you could set its **triggerEvent** to **change**, which will have the effect of performing validation on **emailTI** each time a new character is entered into that field. See Example 10-10.

Example 10-10. Using the **triggerEvent** property to validate input in real time

```
<mx:EmailValidator id="emailValidator"
  source="{emailTI}" property="text"
  triggerEvent="change"/>
```

This is a nice concept to help understand triggers, but this code wouldn't create a particularly user-friendly experience, because users would see error messages as they were typing. We can make the trigger work better by linking it to a submit button. Let's create that **Button** to support the example.

Create the submit button by dragging a **Button** control to the **Form**. Once again, Design mode will wrap the **Button** in a **FormItem** container. The **FormItem** helps align the **Button**, but it doesn't make sense to use the **FormItem** control's **label**, because the **Button** already has one, so change the **FormItem** control's **label** to an empty string. While you're at it, give the **Button** an **id** of **submitButton** and the **label** "Submit".

Now you'll change **EmailValidator** so that its **trigger** property points to the submit button, as shown in Example 10-11. Because you want validation to fire when the submit button is clicked, set **triggerEvent** to **click**.

Example 10-11. Combining the **trigger** and **triggerEvent** properties to control when the **EmailValidator** performs validation

```
<mx:EmailValidator id="emailValidator"
  source="{emailTI}" property="text"
  trigger="{submitButton}"
  triggerEvent="click"/>
```

Because you have overridden the default behavior of the validator, validation will no longer occur when focusing out of the field; rather, it will occur only when the `submitButton` is clicked.

The `validate()` method

If you want even more control over when validation occurs, you can use a validator's `validate()` method to trigger the validation at any time. To test this, first remove any changes you've made to the `trigger` and `triggerEvent` properties in your code. Then, create a function that calls the `validate()` method for the `EmailValidator`, shown in Example 10-12.

Example 10-12. Calling a validator's `validate()` method from a function

```
private function validateAndSubmit():void{
    emailValidator.validate();
}
```

Finally, set the `click` event on `submitButton` to call this function. With this code in place, clicking `submitButton` will trigger an additional validation on `emailTI`. Since the validator has a `source` and `property` assignment, it will continue to validate when focus is changed, but this gives you a second *modus operandi* when it comes to triggering validation.

Linking several validators with `Validator.validateAll()`

If you're imagining situations where you might want to call `validate()` on every validator within a single function, stop right there, because you can use the static method `validateAll()` on the `mx.validators.Validator` class. The term *static method* means the method is called directly from a class, not an instance of the class. In this case, we'll import the `Validator` class and call `validateAll()`, which takes an `Array` of validators as its parameter, directly from the class. Example 10-13 demonstrates how to revise the `validateAndSubmit()` function to set up and call `validateAll()`.

Example 10-13. Using the static method `validateAll()` to trigger validation on every validator component in the application

```
import mx.validators.Validator;

private function validateAndSubmit():void{
    var validatorsArray:Array;
    validatorsArray = [firstNameValidator, lastNameValidator,
        emailValidator, phoneValidator, zipCodeValidator];
    Validator.validateAll(validatorsArray);
}
```

Now validation will trigger for all these validators at once when the `submitButton` is clicked.

NOTE

Remember to place `ActionScript` functions within the `Script/CDATA` block. Also, don't forget to configure `submitButton` to call `validateAndSubmit()`, which you can do like so:

```
<s:Button id="submitButton"
    label="Submit"
    click="validateAndSubmit()"/>
```

WARNING

Flash Builder's code completion has a hard time using dot notation to sense static methods on classes that have not yet been imported, but you can overcome this problem and force code hinting (and auto-importing) by typing half of the class name—in this case, `valid`—and then triggering code completion with a keyboard shortcut. For Windows, trigger code hinting using `Control+space bar`; for Mac OS, trigger code hinting using `Command+Shift+space bar`.

NOTE

Like the `Validator` class and its method `validateAll()`, the `show()` method on the `Alert` class is another example of a static method.

At this point, when someone clicks Submit, she probably expects everything to be fine with her form. If it's not, the red borders around incorrect fields might not be as obvious as you'd like. If this is your feeling, you can always use **Alert** to create a message, grab the user's attention, and focus her on the problems detected.

To show an **Alert**, simply call the static **show()** method on the **Alert** class. We used this technique to announce "game over" in Chapter 7 when we created the *Collision!* game. As you may remember, **Alert.show()** takes two parameters, the first being the body of the **Alert**, and the second being the title of the **Alert** window. So, after importing **mx.controls.Alert** (or using the trick we described in the warning note about code completion), you can append the **validateAndSubmit()** function to produce an error message, as demonstrated in Example 10-14.

Example 10-14. Adding an error message following validation

```
private function validateAndSubmit():void{
    var validatorsArray:Array;
    validatorsArray = [firstNameValidator, lastNameValidator,
        emailValidator, phoneValidator, zipCodeValidator];
    Validator.validateAll(validatorsArray);

    Alert.show("Please fix that stuff.",
        "There were problems with your form.");
}
```

If you run your application and test this function, though, you'll discover a problem: we still need to confirm the form truly has errors before we post the error message. Fortunately, **Validator.validateAll()** returns an **Array** of results, and because arrays have a **length** property noting the number of items they contain, checking for a **length** greater than zero (**array.length > 0**) is the best way to test whether the validation components returned any errors. We'll handle this test using an **If..Else If** block.

The revision provided in Example 10-15 will run one of two code branches depending on which condition is **true**. If **errorsArray** has a **length** of 1 or more, the error message will post; otherwise, if **errorsArray** has a **length** of 0, a success message will post.

Example 10-15. Testing for validation problems before posting an error message

```
private function validateAndSubmit():void{
    var validatorsArray:Array;
    var errorsArray:Array;

    validatorsArray = [firstNameValidator, lastNameValidator,
        emailValidator, phoneValidator, zipCodeValidator];

    errorsArray = Validator.validateAll(validatorsArray);

    if(errorsArray.length > 0){
        Alert.show("Please fix that stuff.",
            "There are problems with your submission");
    }
```



```

    }else if(errorsArray.length == 0){
        Alert.show("The information was submitted",
            "Rich Forms Application");
    }
}

```

We'll add one more level of complexity to the function before we call it "done." Instead of displaying a generic **Alert** indicating that some unspecified fields have problems, we'll display the actual errors registered by **validateAll()**.

To accomplish this, we need to learn one more scripting technique: *looping*. Looping lets you run a piece of code repeatedly until certain conditions are met. One type of loop is the **For..Each..In** loop. As you likely imagine, this loop runs a block of code for each item in an **Array** (or an object, etc.).

When an **Array** of errors is returned by **validateAll()**, each object returned, which is a **ValidationResultEvent** class, will include a **message** property as a **String** variable, and each **message** will correspond to an actual error registered by a specific validator. Getting to the error message is the easy part. The "trick" lies in not only finding the control that threw the error, but also referencing that control in an error message using a format the user will recognize. In other words, we want to refer to "Email" and not something cryptic like **emailTI**. We'll accomplish our goal by using a **For..Each..In** loop to gradually build a **String** that we'll use as a composite error message. Example 10-16 demonstrates how we do this.

Example 10-16. Building a composite error message using a *For..Each..In* loop

```

private function validateAndSubmit():void{
    var validatorsArray:Array;
    var errorsArray:Array;
    var errorString:String = "";

    validatorsArray = [firstNameValidator, lastNameValidator,
        emailValidator, phoneValidator, zipCodeValidator];

    errorsArray = Validator.validateAll(validatorsArray);

    if(errorsArray.length > 0){

        for each (var error:ValidationResultEvent in errorsArray){
            var formItem:FormItem = FormItem(error.target.source.parent);

            errorString += formItem.label;
            errorString += ": " + error.message + "\n\n";
        }

        Alert.show(errorString,
            "There are problems with your submission");
    }else if(errorsArray.length == 0){
        Alert.show("The information was submitted",
            "Rich Forms Application");
    }
}

```

Pay close attention to the **For...Each...In** loop. The opening **for each** statement creates an **error** variable as an instance of the next **ValidationResultEvent** object in **errorsArray**, which we're deconstructing. Once inside the loop, the first line casts/recognizes the value of **error.target.source.parent** as a **FormItem** and stores it as a local variable; in other words, we're casting an unknown parent as a **FormItem** so we can access its properties, specifically its **label**. The second line adds the **FormItem** component's **label** property to the **errorString** using the addition assignment operator (**+=**). The third line follows by appending the growing error string with the actual error message and a couple of line breaks using the newline escape sequence (**\n**). Finally, the process is repeated *for each* item in the **Array** of **ValidationResultEvent** objects. Once the process completes, the **errorString** is served to **Alert.show()** as the **message** parameter. Between the **For...Each** loop construction, the casting example, and the additive concatenation, there's a lot to learn from this loop!

Now the form is looking pretty solid, as shown in Figure 10-11. In the next section, we use another technique to limit the kinds of values that can be entered in a **TextInput** control.

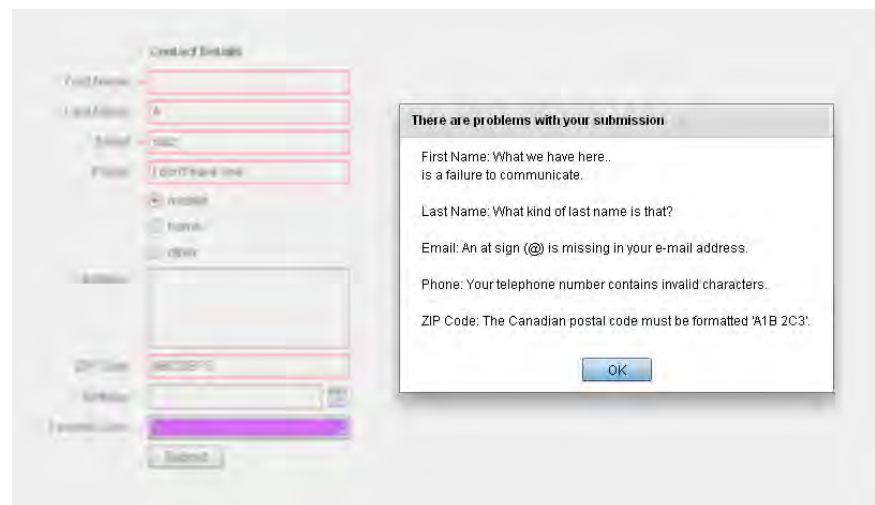


Figure 10-11. An advanced validation scheme

Restricting Input

We don't know anyone with a name like `7*%32$`, so why should we allow someone to enter such characters for a name? Well, you can *restrict* allowable characters for a **TextInput** and prevent people from entering certain numbers or symbols in a field. You accomplish this using an input control's **restrict** property. Modifying **firstNameTI** as shown in Example 10-17 will allow only lowercase letters, uppercase letters, and spaces to be entered as input; other characters will be denied.

Example 10-17. Putting restrictions on user input with the `restrict` property

```
<s:TextInput id="firstNameTI" width="188"
  restrict="a-zA-Z" />
```

However, you've missed something. What if someone has a hyphenated first or last name, such as Day-Lewis, or a name that includes an apostrophe, such as O'Reilly? You'll need to add a few more characters to the `restrict` property. The problem is, the hyphen is a special character used by the property to show a range, so how do you tell Flash Builder you definitely want a hyphen? Once again, we'll use an escape sequence (`\`). Improve the `restrict` property by following the space with an apostrophe, and then add a backslash preceding a hyphen to show that you want to include the hyphen literally. Example 10-18 shows the syntax.

Example 10-18. A more advanced `restrict` property requiring an escape character

```
<s:TextInput id="firstNameTI" width="188"
  restrict="a-zA-Z '\-' />
```

While you're at it, apply the `restrict` property to the First Name and Last Name fields to prevent people from entering unwanted characters.

You can use this same idea for the Phone field if you want, restricting it to just numbers, parentheses, hyphens, and so on. You also might want to craft a `restrict` property for the ZIP Code field. Canadian postal codes use letters as well as numbers, though, so keep that in mind. Example 10-19 shows the restriction we used for `phoneTI`.

Example 10-19. A `restrict` property limiting input into the Phone field

```
<s:TextInput id="phoneTI" width="188"
  restrict="0-9 .()\-"/>
```

Formatting Input

Flex comes with some formatter components that allow you to automatically sculpt the most common types of data. Using *formatters*, you can keep your data in a raw form and modify it as necessary. Say you have prices of products stored away in a database or an XML file. It's nice to keep them stored as numbers so you can manipulate them (adding discounts, changing currencies, and so on), but you also want to show the prices as dollars for people using your application—complete with a dollar sign, comma separators for the thousands place, and rounded to two decimal places. After all, this is how people expect to see prices.

Formatters take care of all this. Formatters exist for the most common types of data, such as currencies, dates, numbers, phone numbers, and zip codes. Like validators, formatters are nonvisual components, and as such they should be added within the Declarations section of your code. To use

NOTE

Restricting allowable characters in a `TextInput` also prevents special characters from being used, such as those in non-English languages. To get around this, instead of specifying what characters you will allow, use the inverse functionality of the `restrict` property to specify only characters you do not want. Do this by prepending the string sequence with a caret (^).

For example, to exclude only numbers from a `TextInput`, change the property to `restrict="^0-9"`, allowing entry of anything except 0–9. This is actually regular expression syntax.

NOTE

You might be curious why we haven't demonstrated methods to auto-format fields such as `phoneTI` as you type. For instance, inputting into the phone number field could automatically add parentheses to the area code or add hyphens after the first three digits of the phone number. That's generally referred to as masking an input field. Out of the box, Flex doesn't provide stock masking components, but some third-party sources likely provide such functionality. Perhaps you could also use this thought as inspiration for your own custom component once you complete this chapter.

a formatter, simply create a tag for the formatter you want, and then call its `format()` method when you want to process some data. The `format()` method returns a string of text.

You'll get your hands dirty with formatters in the next section, in which you'll create a short and sweet application demonstrating how restrictions and formatters work together. For the time being, just gloss over the following background material in order to grasp the basic concepts.

Sticking with the currency scenario, we would first need a **CurrencyFormatter** tag in the Declarations section of an application, as shown in Example 10-20.

Example 10-20. *A CurrencyFormatter in the Declarations section*

```
<fx:Declarations>
  <mx:CurrencyFormatter id="currFormatter"/>
</fx:Declarations>
```

Now that we have a formatter component, we need to call its `format()` method and pass it a value. For example, `currFormatter.format(10243)` would return the string "\$10,243" with both the dollar sign (\$) and comma added. The comma separator is a default behavior of the **CurrencyFormatter**, but if you don't like it, you can change the property `useThousandsSeparator` to `false`.

What's more, it's simple to use a formatter in concert with inline script. Consider Example 10-21, which uses an inline expression to run the formatter's `format()` method on a **TextInput** control's `text` property.

Example 10-21. *Using inline script to call a format() method*

```
<fx:Declarations>
  <mx:CurrencyFormatter id="currFormatter"/>
</fx:Declarations>

<s:TextInput id="priceTI" text="{currFormatter.format(priceTI.text)}/>
```

Of course, **CurrencyFormatter** provides other useful properties; one of them is the option to round up or down, and another establishes how many decimal places should display (`precision`).

So, what if a price is in another denomination, such as euros, and you want to round up to the nearest cent? These format options are easy to impose if you know which properties to modify. In this case, demonstrated in Example 10-22, you'll need to tell the formatter to round up to the nearest whole number, display two decimal places, and change the `currencySymbol` from its default (\$) to euros (€).

Example 10-22. *Configuring a formatter to return special results*

```
<mx:CurrencyFormatter id="currFormatter"
  rounding="up" precision="2" currencySymbol="€"/>
```

In this case, submitting the value 10243.5678, would return "€10,243.57".

Besides for currency, formatters exist for dates, phone numbers, zip codes, and other types of form information. For instance, you could use a **PhoneFormatter** tag to format a given value into a phone number, as shown in Example 10-23. By default, the **PhoneFormatter** returns 10-digit numbers as the familiar (###) ###-####.

Example 10-23. *Configuring and calling a PhoneFormatter*

```
<fx:Declarations>
  <mx:PhoneFormatter id="defaultPhoneFormat"/>
</fx:Declarations>

<s:TextInput text="{defaultPhoneFormat.format('4795558273')}" />
```

This combination would output “(479) 555-8273”.

But, how could you tweak the **PhoneFormatter** to produce a custom format? It's easy; just set the **formatString** property. The **PhoneFormatter** replaces pound symbols (#) in its **formatString** with numeric characters from the supplied input, separated by whatever delimiters are used in the **formatString**. Example 10-24 demonstrates a **PhoneFormatter** that uses periods as delimiters.

Example 10-24. *Configuring a PhoneFormatter to render custom output by applying its formatString property*

```
<fx:Declarations>
  <mx:PhoneFormatter id="phoneFormatter" formatString="###.###.###"/>
</fx:Declarations>

<s:TextInput text="{phoneFormatter.format('4795558273')}" />
```

This version would return “479.555.8273”.

Combining Restrictions and Formatters

Example 10-25 contains a quick demo called **FormatApp** that combines restrictions and formatters; see the result in Figure 10-12. It makes use of the **CurrencyFormatter** and the **PhoneFormatter** components we just discussed, but notice how each formatter's counterpart **TextInput** field also restricts input to yield clean, formatted values. For instance, **priceTI** is restricted solely to number characters. Similarly, **phoneTI** limits input to numbers and a maximum of 10 digits. Write it and run it, and of course, test its reaction to some awkward input.

Example 10-25. *A demo application combining restrictions and formatters*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">
```

Formatting Demo:

Currency:

€221.59

Phone:

479.555.8273

Figure 10-12. Experimenting with restrictions and formatters

```

<fx:Declarations>
  <mx:CurrencyFormatter id="currFormatter"
    rounding="up" precision="2" currencySymbol="€"/>
  <mx:PhoneFormatter id="phoneFormatter"
    formatString="###.###.####"/>
</fx:Declarations>

<s:VGroup horizontalCenter="0" verticalCenter="0" width="300">

  <s:Label text="Formatting Demo:" fontWeight="bold"/>

  <s:Label text="Currency:"/>
  <s:HGroup verticalAlign="middle">
    <s:TextInput id="priceTI" restrict="0-9."/>
    <s:Label id="priceLabel" paddingLeft="25"
      text="{currFormatter.format(priceTI.text)}/>
  </s:HGroup>

  <s:Label text="Phone:"/>
  <s:HGroup verticalAlign="middle">
    <s:TextInput id="phoneTI" restrict="0-9" maxChars="10"/>
    <s:Label id="phoneLabel" paddingLeft="25"
      text="{phoneFormatter.format(phoneTI.text)}/>
  </s:HGroup>

</s:VGroup>

</s:Application>

```

Linking Formatters to Functions

For this chapter's final lesson, we return to the **RichForms** application we created in the first section. This time, though, we'll add a **Panel** container and feed it some formatted output. The purpose of the lesson is to demonstrate linkage between the MXML formatter components and ActionScript functions.

In the Declarations section of the **RichForms** application code, add the formatter components shown in Example 10-26 below the validator tags. If you didn't create or didn't save the validators from the prior lesson, you may opt to create them now, but they are not required for the example to work.

Example 10-26. Adding formatters to the *RichForms* application

```

<fx:Declarations>
  <s:RadioButtonGroup id="phoneRadioButtonGroup"/>

  <!-- validators are here -->

  <mx:PhoneFormatter id="phoneFormatter" formatString="###.###.####"/>

  <mx>DateFormatter id="dateFormatter"/>
</fx:Declarations>

```

Next, create an **HGroup** container with **verticalAlign="middle"** and copy the code for the **Form** into the container. Then, add the **Panel** code from Example 10-27 below the **Form**.

Example 10-27. Adding a details preview pane to the RichForms application

```
<s:HGroup verticalAlign="middle">

    <!--

        Form code goes here

    -->

    <s:Panel id="detailsPanel" title="Contact Details:"
width="180" height="310" dropShadowVisible="false">
    <s:VGroup horizontalCenter="0" verticalCenter="0"
left="10" right="10" top="10" bottom="10">
    <s:Label id="nameLabel"
        text="Name: {firstNameTI.text} {lastNameTI.text}"/>
    <s:Label id="phoneTypeLabel"
        text="{phoneRadioButtonGroup.selectedValue} Phone Number:"/>
    <s:Label id="phoneLabel"
        text="{phoneFormatter.format(phoneTI.text)}/>
    <s:Label id="birthdayLabel"
        text="Birthday
            {dateFormatter.format(dateField.selectedDate)}/>
    <s:Label id="ageLabel"
        text=""/>
    <s:Label id="colorLabel"
        text="Favorite Color:"/>
    <s:Rect height="50" width="50">
    <s:fill>
    <s:SolidColor color="{colorPicker.selectedColor}"/>
    </s:fill>
    </s:Rect>
    </s:VGroup>
    </s:Panel>

</s:HGroup>
```

The **Panel** code provides a couple of points for discussion. Note the use of the **selectedValue** property on **phoneRadioButtonGroup**. The **RadioButtonGroup** component's **selectedValue** property equals the **label** of the selected **RadioButton**, which can be either “mobile”, “home”, or “other” in this case. Also note the binding of the **birthdayLabel** control's **text** property to the **DateField** control's **selectedDate** property. The **DateFormatter** is passed the **selectedDate** value, and it returns a string with a nicely formatted date. Without this, the default presentation of the **selectedDate** would be a fairly cryptic-looking string, complete with time values. Finally, at the bottom, notice the inclusion of the FXG **Rect** graphic object, complete with nested **Fill** and **SolidColor** properties. We needed to display the favorite color selection somehow, and the FXG graphic element provided the best solution.

For the next phase, we'll add an `ActionScript` function to calculate the contact's age. We'll do this using some methods of the `Date` class. Take a look at the function in Example 10-28, which takes a `Date` object as a parameter and returns age as a `String`.

NOTE

A `Date` object's `time` property is actually the number of milliseconds since midnight on January 1, 1970. This is how `Date` objects store their value internally.

Example 10-28. This function takes a `Date` parameter and returns an age calculation

```
private function calculateAge(birthDate:Date):String{
    var today:Date = new Date();
    var ageDate:Date = new Date(today.time - birthDate.time);
    var age:Number = ageDate.fullYear - 1970;
    return age.toString();
}
```

This function creates a new `Date`, which defaults to the current date and time. To get the person's age, the difference between the date of birth and the current date is calculated. Note that you can't subtract one `Date` from another. However, `Date` does provide a `time` property, which is a numerical representation of a date on which you can perform calculations. So, in order to calculate the difference between today and the birth date, you take the `time` value of the birth date and subtract it from the `time` value of today's date. That value is then converted into the number of years since birth by using the `fullYear` property. Finally, because the `Date` object's standard point of reference is 1970, 1970 must be subtracted to get the final age.

To make use of this function, use an inline script to bind the function's result to `ageLabel`. The inline script will need to pass the `DateField` control's `selectedDate` to the `calculateAge()` function and append the result with the string "years old". When the `DateField` control's `selectedDate` changes, the `Label` will update with the calculated age. Example 10-29 shows how to revise `ageLabel` in order to output age.

Example 10-29. Revising `ageLabel` to output the contact's age by calling `calculateAge()` with a binding expression

```
<s:Label id="ageLabel"
    text="{calculateAge(dateField.selectedDate)} years old"/>
```

Initially, `ageLabel` will have an empty text property. It will not display anything until a value has been selected in the `DateField`. Flex realizes no date is selected on the `DateField`, so it doesn't process the binding. This is great because the `Label` won't display the odd text "years old" until the `DateField` has a value. Any time the `DateField` changes, the function will be called and the binding will update.

When you add this code to your application and run it, you might notice something strange. Specifically, the calculated age might be a fractional number (with decimal places) and a long age value might display, such as "28.662405 years old". That's hardly what you want. Again, formatters come to the rescue. The final addition to this application adds a `NumberFormatter`

with a **rounding** property. This revision will return age values rounded down to the nearest whole number. To include the **NumberFormatter**, add the tag shown in Example 10-30 to your Declarations section.

Example 10-30. Adding a **NumberFormatter** to the **RichForms** application

```
<mx:NumberFormatter id="numberFormatter" rounding="down"/>
```

To use the **NumberFormatter**, call its **format()** method from within the **calculateAge()** function just prior to returning the **age** value as a **String**. Note that since we worked with **age** as a **Number** data type, we have to cast the **String** result returning from **NumberFormatter** as a **Number** in order to receive it with the same variable instance of **age**, as demonstrated in Example 10-31.

Example 10-31. Appending the **calculateAge()** function to call the **NumberFormatter**'s **format()** method

```
private function calculateAge(birthDate:Date):String{
    var today:Date = new Date();
    var ageDate:Date = new Date(today.time - birthDate.time);
    var age:Number = ageDate.fullYear - 1970;
    age = Number(numberFormatter.format(age));
    return age.toString();
}
```

Now, the age displays perfectly. Figure 10-13 shows an example of how your application will look when running.

Contact Details

First Name *

Last Name

Email *

Phone

☐ mobile
☒ home
☐ other

Address

ZIP Code

Birthday

Favorite Color

Contact Details:


Name: Stazmo Abernathy
home Phone Number:
479.555.8273
Birthday: 11/05/1955
54 years old
Favorite Color:


Figure 10-13. The finished contact editor with sample input

The final code for the **RichForms** application is provided in Example 10-32.

NOTE

The **Math** class has a number of methods for performing rounding and calculations. The method **Math.floor()** can accomplish the same effect as rounding down with the **NumberFormatter** component.

Example 10-32. *The completed application code for RichForms*

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
  </s:layout>

  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      import mx.events.ValidationResultEvent;
      import mx.validators.Validator;
      import mx.validators.ZipCodeValidatorDomainType;

      private function onColorSelection():void{
        if(colorPicker.selectedColor == 10027263){
          colorPicker.errorString = "Hmm.. " +
            "not the color we expected.";
        }else{
          colorPicker.errorString = "";
        }
      }

      private function validateAndSubmit():void{
        var validatorsArray:Array;
        var errorsArray:Array;
        var errorString:String = "";

        validatorsArray = [firstNameValidator, lastNameValidator,
          emailValidator, phoneValidator, zipCodeValidator];

        errorsArray = Validator.validateAll(validatorsArray);

        if(errorsArray.length > 0){
          for each (var error:ValidationResultEvent in errorsArray){
            var formItem:FormItem =
              FormItem(error.target.source.parent);
            errorString += formItem.label;
            errorString += ": " + error.message + "\n\n";
          }
          Alert.show(errorString,
            "There are problems with your submission");
        }else if(errorsArray.length == 0){
          Alert.show("The information was submitted",
            "Rich Forms Application");
        }
      }

      private function calculateAge(birthDate:Date):String{
        var today:Date = new Date();
        var ageDate:Date = new Date(today.time - birthDate.time);
        var age:Number = ageDate.fullYear - 1970;
        age = Number(numberFormatter.format(age));
        return age.toString();
      }
    ]]>
  </fx:Script>

```

```
<fx:Declarations>
```

```
<s:RadioButtonGroup id="phoneRadioButtonGroup"/>

<mx:StringValidator id="firstNameValidator"
    source="{firstNameTI}" property="text"
    requiredFieldError="{ 'What we have here..\n'
        + 'is a failure to communicate.' }"/>

<mx:StringValidator id="lastNameValidator"
    source="{lastNameTI}" property="text"
    required="false" minLength="2"
    tooShortError="What kind of last name is that?"/>

<mx:EmailValidator id="emailValidator"
    source="{emailTI}" property="text"/>

<mx:PhoneNumberValidator id="phoneValidator"
    source="{phoneTI}" property="text" required="false"/>

<mx:ZipCodeValidator id="zipCodeValidator"
    source="{zipCodeTI}" property="text" required="false"
    domain="{ZipCodeValidatorDomainType.US_OR_CANADA}"/>

<mx:PhoneFormatter id="phoneFormatter" formatString="###.###.####"/>

<mx>DateFormatter id="dateFormatter"/>

<mx:NumberFormatter id="numberFormatter" rounding="down"/>
```

```
</fx:Declarations>
```

```
<s:HGroup verticalAlign="middle">
```

```
<mx:Form width="324" id="contactForm">
    <mx:FormHeading label="Contact Details"/>
    <mx:FormItem label="First Name" required="true">
        <s:TextInput id="firstNameTI" width="188"
            restrict="a-zA-Z '\-' />
    </mx:FormItem>
    <mx:FormItem label="Last Name">
        <s:TextInput id="lastNameTI" width="188"
            restrict="a-zA-Z '\-' />
    </mx:FormItem>
    <mx:FormItem label="Email" required="true">
        <s:TextInput id="emailTI" width="188"/>
    </mx:FormItem>
    <mx:FormItem label="Phone">
        <s:TextInput id="phoneTI" width="188"
            restrict="0-9 .() \-' />
        <s:RadioButton label="mobile" groupName="phoneRadioButtonGroup"
            selected="true"/>
        <s:RadioButton label="home" groupName="phoneRadioButtonGroup"/>
        <s:RadioButton label="other" groupName="phoneRadioButtonGroup"/>
    </mx:FormItem>

    <mx:FormItem label="Address">
        <s:TextArea height="75"/>
    </mx:FormItem>
```

```

<mx:FormItem label="ZIP Code">
  <s:TextInput id="zipCodeTI" width="188"/>
</mx:FormItem>

<mx:FormItem label="Birthday">
  <mx:DateField id="dateField" width="188"
    yearNavigationEnabled="true"/>
</mx:FormItem>

<mx:FormItem label="Favorite Color">
  <mx:ColorPicker width="188" id="colorPicker"
    change="onColorSelection()"/>
</mx:FormItem>

<mx:FormItem label="">
  <s:Button id="submitButton" label="Submit"
    click="validateAndSubmit()"/>
</mx:FormItem>
</mx:Form>

<s:Panel id="detailsPanel" title="Contact Details:"
  width="180" height="310" dropShadowVisible="false">
  <s:VGroup horizontalCenter="0" verticalCenter="0"
    left="10" right="10" top="10" bottom="10">

    <s:Label id="nameLabel"
      text="Name: {firstNameTI.text} {lastNameTI.text}"/>

    <s:Label id="phoneTypeLabel"
      text="{phoneRadioButtonGroup.selectedValue} Phone Number:"/>

    <s:Label id="phoneLabel"
      text="{phoneFormatter.format(phoneTI.text)}"/>

    <s:Label id="birthdayLabel"
      text="Birthday:
        {dateFormatter.format(dateField.selectedDate)}"/>

    <s:Label id="ageLabel"
      text="{calculateAge(dateField.selectedDate)} years old"/>

    <s:Label id="colorLabel"
      text="Favorite Color:"/>

    <s:Rect height="50" width="50">
      <s:fill>
        <s:SolidColor color="{colorPicker.selectedColor}"/>
      </s:fill>
    </s:Rect>
  </s:VGroup>
</s:Panel>

</s:HGroup>

</s:Application>

```

Summary

In this chapter, you learned how to use validation techniques, from the very simple to the fairly advanced. You learned to use a variety of validators and even created a robust validation scheme in ActionScript. You learned how to restrict input fields, limiting the type of characters that are allowed, and you learned to use a few new controls, such as **RadioButton** and **DateField**.

This chapter also covered looping, which is a more advanced scripting technique than we have used in previous exercises. You learned how to format data by creating an application using some formatter classes. You also learned to perform calculations on dates and times. All of this, used in conjunction with the form layout, gives you the know-how to create rich, user-friendly forms for Flex applications.

GATHERING AND DISPLAYING DATA

“Make a list of things you need. Throw it out, do what you feel.”

—Nick Vahle,
Time to Shine

Few applications are complete without using some form of data. This is especially true with web applications. Just think of the web applications you use often. How many of them would serve you any purpose if they did not access some form of data? It is the ability to seamlessly connect to information that has made the Web such a significant medium of communication and expression.

Whether the information you send and receive is stored in an XML file, stored in a database, or gathered from one of the many web services available, you can be sure that with Flex, gathering and displaying data is fairly straightforward.

Using List-Based Controls

Flex offers a number of controls known as *List-based controls*, which make displaying a list of items very easy. All **List**-based controls have the ability to accept either a simple list of data or complex, structured data, and they offer a number of customizable features. Common controls implementing list functionality include the following:

List

This is the backbone of all list controls; it orders its items vertically.

DropDownList

The **DropDownList** control, which renders display items using the **Label** component, allows the user selection of a predefined set of options.

IN THIS CHAPTER

- Using List-Based Controls
- Using XML Data
- Implementing List Selection
- Connecting to Search Results
- Dragging and Dropping in Lists
- Creating Custom Item Renderers
- Working With External Data Services
- Summary

ComboBox

The **ComboBox** is similar to the **DropDownList** in that it provides a selection of items. However, the **ComboBox** renders display against the **TextInput** and, as such, is editable.

ButtonBar

The Spark **ButtonBar** allows for similarly styled buttons to be arranged either horizontally or vertically. Support of custom item rendering allows for quick integration of icons, rather than settling for the default **Button** appearance.

HorizontalList (*Halo only*)

One of the Halo holdovers, **HorizontalList** controls arrange their items horizontally. They are commonly used in conjunction with e-commerce sites and photo galleries to display thumbnails of various imagery.

DataGrid (*Halo only*)

The **DataGrid** is an advanced list control that can display multiple columns of data arranged in a table. Rows can be sorted by field headings, and columns can be resized and even rearranged by dragging and dropping content.

AdvancedDataGrid (*Halo only*)

The **AdvancedDataGrid** is similar to the regular **DataGrid**, but it offers more functionality, including the ability to sort by multiple columns.

Lists of Simple Data

The UI controls introduced in the previous list require a **dataProvider** object to supply their content. Spark components require data to be organized as lists or collections; the **ArrayCollection**, **ArrayList**, and **XMLListCollection** classes meet this need. In addition to the previously mentioned data objects, Halo components can also take basic **Array** and **XML** objects as **dataProvider** types.

The first data sequence you'll use in this chapter is a list of color names, and you'll store them as an **ArrayCollection**.

You can create an **ArrayCollection** in MXML using the `<s:ArrayCollection/>` tag in conjunction with nested `<fx:String/>` tags. Example 11-1 demonstrates how to create an **ArrayCollection** using MXML.

NOTE

*Using the **ArrayCollection** over the **Array** provides a particular advantage: the **ArrayCollection** class listens for additions and removals of items to and from the underlying source data. This value-add distinguishes the **ArrayCollection** from the **Array** for use in dynamic applications with frequently changing values.*

Example 11-1. *An ArrayCollection data object created with MXML*

```

<s:ArrayCollection>
  <fx:String>Red</fx:String>
  <fx:String>Green</fx:String>
  <fx:String>Blue</fx:String>
</s:ArrayCollection>

```

In ActionScript, you create an **ArrayCollection** by first building an **Array** and then attaching that **Array** to an **ArrayCollection** as a **source**, as shown in Example 11-2.

Example 11-2. *Creating an Array as a source for an ArrayCollection using ActionScript*

```

var array:Array = ["Red", "Green", "Blue"];
var arrayCollection:ArrayCollection = new ArrayCollection();
arrayCollection.source = array;

```

However, there is a shorthand approach to creating an **ArrayCollection**. Example 11-3 gets the same result by casting raw data served in array syntax as an **ArrayCollection** object at the moment of initialization.

Example 11-3. *Creating an ArrayCollection in ActionScript using a shorthand approach*

```

var ac:ArrayCollection = new ArrayCollection(["Red", "Green", "Blue"]);

```

Now that you know how to create simple lists of data, let's display that data in a **List** control. To do that, you'll feed the **ArrayCollection** to a **dataProvider** property of a **List**.

Continuing with our original example of the MXML **ArrayCollection**, one method of filling a **List** with these three colors would be to build the **dataProvider** and **ArrayCollection** as nested tags within the **List** control itself, as in Example 11-4.

Example 11-4. *Populating a List control with a nested MXML data object*

```

<s>List id="list">
  <s:dataProvider>
    <s:ArrayCollection>
      <fx:String>Red</fx:String>
      <fx:String>Green</fx:String>
      <fx:String>Blue</fx:String>
    </s:ArrayCollection>
  </s:dataProvider>
</s>List>

```

Such a tag-based **dataProvider** is fine for short lists of values, but it becomes a monstrosity in your code when it's larger or when you have several similar components. For these reasons, you might prefer to arrange your **dataProvider** exclusively in ActionScript.

Example 11-5, which uses `ActionScript`, creates an **Array** and then establishes the **Array** as the source of the **ArrayCollection**. Once the collection object is built, it's assigned to the **List** control's **dataProvider** property. The function is called by the **List** control's **creationComplete** event.

Example 11-5. *Populating a List control with ActionScript*

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <fx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      private function loadData():void{
        var array:Array = ["Red", "Green", "Blue"];
        var arrayCollection:ArrayCollection = new ArrayCollection();
        arrayCollection.source = array;
        list.dataProvider = arrayCollection;
      }
    ]]>
  </fx:Script>

  <s:List id="list" creationComplete="loadData()"/>

</s:Application>
```

NOTE

The **dataProvider** is the property to remember for any Flex component that absorbs a list of items.

Yet another option remains. At times, you might find it convenient to declare the entire **ArrayCollection** object inside the `</fx:Declarations>` block and then bind it to a **List** control's **dataProvider**, as in Example 11-6.

Example 11-6. *Populating a List control by binding its dataProvider to an ArrayCollection in the Declarations section*

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <fx:Declarations>
    <s:ArrayCollection id="arrayCollection" source="[Red, Green, Blue]"/>
  </fx:Declarations>

  <s:List dataProvider="{arrayCollection}"/>

</s:Application>
```

Arrays and Lists in Flex 3

The Flex 4 **List** controls are “more particular” than their Flex 3 counterparts. Specifically, the Flex 3 list will accept a basic **Array** as its **dataProvider**. Of course, it coerces the basic **Array** into an **ArrayCollection** as needed, but it does work. You can demonstrate this for yourself in Flash Builder using the following code:

```
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            private var ar:Array = ["Red", "Green", "Blue"];
        ]]>
    </fx:Script>

    <mx:List id="myList" dataProvider="{ar}"/>

</s:Application>
```

Inside a script block, this code establishes an **Array** variable with local scope. Elsewhere within the MXML, a **List** component’s **dataProvider** property is bound to the **Array**. Notice that the **List** component comes from the Halo namespace (**mx:**). Run it like this, and it works. But if you change the **List** control’s namespace to Spark (**s:**), Flash Builder will ask you to reconsider.

At some point—if you haven’t already—you’ll turn to Internet searches with a code question, so you should be aware of this difference. Because Flex surged in popularity with the release of Flex 3, you will frequently encounter Flex 3 examples. Many Flex 3 examples will transfer smoothly into Flex 4, but as you just witnessed, some won’t.

Consequently, don’t be surprised if we occasionally refer to “an array of” this or that when discussing Flex 4 list examples. We actually mean “an appropriate **dataProvider** object with a source array of” this or that.

Lists of Complex Data

In the previous examples, we used the **List** component to absorb an **Array** of strings from an **ArrayCollection**. However, as we shall see, the **List** control can also handle an array of arbitrary object data.

Consider a scenario where you have a list of songs to feed a media player. You could show only the song names, but you might want some additional information, such as the artist or the album name. In that case, you might want to use not an array of strings but an array of objects. Because an object can possess any properties you desire, it makes a convenient container for a grab bag of information. For Example 11-7, the properties **song**, **artist**, and **album** are useful.

Example 11-7. *A List control with a nested data provider*

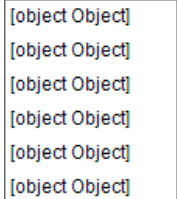
```

<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:List horizontalCenter="0" verticalCenter="0">
    <s:dataProvider>
      <s:ArrayCollection id="musicAC">
        <fx:Object song="Creeper" artist="Phonogenic"
          album="20 Minutes More"/>
        <fx:Object song="Julia" artist="Off the Record"
          album="The Record Store"/>
        <fx:Object song="Forgotten"
          artist="Truett and the Traitors"
          album="The Wind is Gone"/>
        <fx:Object song="Just Wait" artist="Scott and Zamora"
          album="7"/>
        <fx:Object song="Rocket to the Pocket" artist="The Guise"
          album=""/>
        <fx:Object song="So'Co' and Lime" artist="Saturn V"
          album="Extra Solar Sunrise"/>
      </s:ArrayCollection>
    </s:dataProvider>
  </s:List>

</s:Application>

```



[object Object]
[object Object]
[object Object]
[object Object]
[object Object]
[object Object]

Figure 11-1. *A List with no label property*

Now you have a **List** populated by an array of objects, each having unique properties. But take a look at Figure 11-1 to see what this list really looks like.

That's probably not what you wanted. So, what happened? Because we used a **dataProvider** full of objects instead of simple strings, the **List** wasn't sure what to display. Because the regular object has a default string representation of "[object Object]", that's what we got.

Custom label fields

When your data is complex, the **List** control has a property called **labelField** that takes the name of the field/column you want displayed. Since we want to display the song titles, set the **List** control's **labelField** value to "song". Example 11-8 and Figure 11-2 demonstrate the modification and illustrate the result.

Example 11-8. *Assigning the labelField on the List control*

```

<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:List horizontalCenter="0" verticalCenter="0" labelField="song">
    <s:dataProvider>
      <s:ArrayCollection id="musicAC">
        <fx:Object song="Creeper" artist="Phonogenic"
          album="20 Minutes More"/>
      </s:ArrayCollection>
    </s:dataProvider>
  </s:List>

</s:Application>

```

```

    <fx:Object song="Julia" artist="Off the Record"
      album="The Record Store"/>
    <fx:Object song="Forgotten"
      artist="Truett and the Traitors"
      album="The Wind is Gone"/>
    <fx:Object song="Just Wait" artist="Scott and Zamora"
      album="7"/>
    <fx:Object song="Rocket to the Pocket" artist="The Guise"
      album=""/>
    <fx:Object song="So'Co' and Lime" artist="Saturn V"
      album="Extra Solar Sunrise"/>
  </s:ArrayCollection>
</s:dataProvider>
</s:List>

</s:Application>

```

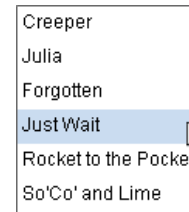


Figure 11-2. A List of song titles

By default, the **List** control's **labelField** looks for a "label" property. So you could accomplish the same thing by changing the property "song" to "label". However, this isn't the recommended solution in this case, as the **List** control provides a graceful workaround.

Advanced lists

What if you wanted to display more than one property at a time? The **DataGrid** was designed for that very purpose. The **DataGrid** lets you specify multiple columns, each mapped to a property or field in the data collection you passed to it. Essentially, it's a table, similar to a spreadsheet or an HTML table.

Flash Builder's Design mode makes it easy to begin a **DataGrid**. Simply drag and drop a **DataGrid** on the stage, and Flash Builder will write the necessary code for you, which you can easily modify to suit your needs. Dropping a **DataGrid** onto the stage should give you the MXML in Example 11-9.

Example 11-9. A default **DataGrid** as created in Design mode

```

<mx:DataGrid>
  <mx:columns>
    <mx:DataGridColumn headerText="Column 1" dataField="col1"/>
    <mx:DataGridColumn headerText="Column 2" dataField="col2"/>
    <mx:DataGridColumn headerText="Column 3" dataField="col3"/>
  </mx:columns>
</mx:DataGrid>

```

The **DataGrid** can display multiple columns, but you have to specify their numbers and traits within a **columns** block, which takes an array of **DataGridColumn**s. So you'll specify which elements from your data object should appear in each individual **DataGridColumn**. Or, said another way, you'll map each column's **dataField** property to some element in your data object. The **headerText** property specifies the column's label. It's optional because it will default to the name of the property attribute specified in the **dataField**. However, you'll often want to modify it to display something more descriptive or text that has different case.

NOTE

DataGrid, **columns**, and **DataGridColumn** are holdovers from the Halo namespace. Unlike their Spark counterparts, **List**-based Halo components can take a basic **Array** as their **dataProvider**. In fact, the **columns** component is really just an extension of the **Array**.

NOTE

You don't have to fully run your project just to compile your code. By merely saving a project, Flash Builder will recompile your code and alert you to any obvious problems. To save your project, choose File→Save. If you're using Windows, use Ctrl+S.

WARNING

The Halo **DataGrid** requires its **dataProvider** complement to come from the Halo (mx:) namespace.

NOTE

The **DataGridColumn** has many customizable properties. We took advantage of the **width** property in Example 11-10.

DataGrids, like all **List** controls, take a **dataProvider** to populate them. So in Example 11-10 we'll reuse our **ArrayCollection** from the previous example to try out the **DataGrid**. To create a **DataGrid** of songs, artists, and albums, copy and paste the **dataProvider** property from the previous example and add it below the closing tag of a **columns** block. But, before you compile the code, move the **dataProvider** to the Halo (mx:) namespace; if you don't, Flash Builder will complain. Figure 11-3 shows the result.

Example 11-10. Assigning field values to the **DataGridColumn** using the **dataField** property

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <mx:DataGrid horizontalCenter="0" verticalCenter="0">
    <mx:columns>
      <mx:DataGridColumn width="130" headerText="Song"
        dataField="song"/>
      <mx:DataGridColumn width="135" headerText="Artist"
        dataField="artist"/>
      <mx:DataGridColumn width="125" headerText="Album"
        dataField="album"/>
    </mx:columns>
    <mx:dataProvider>
      <s:ArrayCollection id="musicAC">
        <fx:Object song="Creeper" artist="Phonogenic"
          album="20 Minutes More"/>
        <fx:Object song="Julia" artist="Off the Record"
          album="The Record Store"/>
        <fx:Object song="Forgotten"
          artist="Truett and the Traitors"
          album="The Wind is Gone"/>
        <fx:Object song="Just Wait" artist="Scott and Zamora"
          album="7"/>
        <fx:Object song="Rocket to the Pocket" artist="The Guise"
          album=""/>
        <fx:Object song="So'Co' and Lime" artist="Saturn V"
          album="Extra Solar Sunrise"/>
      </s:ArrayCollection>
    </mx:dataProvider>
  </mx:DataGrid>

</s:Application>
```

Song	Artist	Album
Creeper	Phonogenic	20 Minutes More
Julia	Off the Record	The Record Store
Forgotten	Truett and the Traitors	The Wind is Gone
Just Wait	Scott and Zamora	7"
Rocket to the Pocket	The Guise	
So'Co' and Lime	Saturn V	Extra Solar Sunrise

Figure 11-3. A **DataGrid** control with multiple fields for song, artist, and album information

“‘Cos I Want You to Know”

We created an example audio player widget complete with a cool frequency spectrum graphic shortly before going to press (Figure 11-4), but unfortunately, we just didn't have enough time to work it into the book as an additional example. However, we released the application with View Source enabled (right-click the widget and select View Source), so there's nothing stopping you from visiting Elijah's home page (<http://www.elrobis.com/>) and downloading the application code, which you can learn from and customize in order to make your own web-based audio player. The applet uses data in an XML file, so there shouldn't be anything overly complicated about it.

Using XML Data

So far, you've seen a few examples demonstrating methods of populating **List** controls with **ArrayCollection** data. But these controls can also handle another type of data: XML.

The XML Structure

We'll demonstrate how to handle XML data in **List** controls by working with some hypothetical contact information. In the XML structure that follows, we start our data object with the `<contacts/>` tag, the lowest level of our XML. Afterward, each unique contact in our data set, represented by the `<contact/>` tag, will be assigned an `id` attribute, as well as first and last name, email address, and phone number child nodes. Here's the structure:

```
<contacts>
  <contact id="">
    <firstName/>
    <lastName/>
    <email/>
    <phone/>
  </contact>
</contacts>
```

The `<contacts/>` tag is the *root* of the XML. The root is the main tag all valid XML requires. Below it, various `<contact/>` nodes unfold. We can establish any number of `<contact/>` nodes.

The structure of the XML affects how we refer to and call its values later. Each `<contact/>` will include an attribute—`id`—that provides a unique numerical identifier for that contact. The other properties for each `<contact/>`, which are the name, email, and phone values, will be handled by child tags. These properties could just as well be expressed as attributes, but it's always a safe bet to use child tags. Later you may decide to add properties to the child tag, or you may want to include other, more complex content in that child tag.

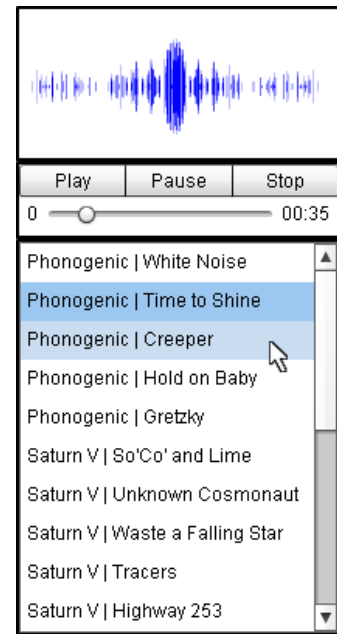


Figure 11-4. A Flex Audio Player widget complete with a spectrum graphic (see the sidebar titled “‘Cos I Want You to Know” above).

NOTE

The `id` attribute going into our XML `<contact/>` tags has no relationship to the `id` attribute for Flex components. This is an arbitrary name you're using as an XML attribute, and you could use any name as a substitute, such as “number”.

You're encouraged to create your own list of contacts using this template. Doing so helps individualize the example as well as stimulate your memory. And besides, you can always add some entertainment value.

The next example demonstrates XML written directly into the Declarations section of an application. XML data is wrapped within the `<fx:XML/>` tag, and the whole object will be placed in the `<fx:Declarations/>` section. Example 11-11 is an XML assortment of contacts we created to run this demo exercise.

Example 11-11. *An XML object created in the Declarations section*

```
<fx:Declarations>
  <fx:XML id="contactsXML" xmlns="">
    <contacts>
      <contact id="1" type="person">
        <firstName>John</firstName>
        <lastName>Jingleheimerschmidt</lastName>
        <email>jakejingleheimerschmidt@gmail.com</email>
        <phone>4795552134</phone>
      </contact>
      <contact id="2" type="person">
        <firstName>Goldie</firstName>
        <lastName>Locks</lastName>
        <email>goldielocks.2010@hotmail.com</email>
        <phone>7015551234</phone>
      </contact>
      <contact id="3" type="business">
        <firstName>Charlestown Chiefs</firstName>
        <lastName/>
        <email>chiefs.hockey@yahoo.com</email>
        <phone>8145554421</phone>
      </contact>
      <contact id="4" type="person">
        <firstName>Red</firstName>
        <lastName>Ridinghood</lastName>
        <email>redridinghood.2010@gmail.com</email>
        <phone>4155552341</phone>
      </contact>
      <contact id="5" type="person">
        <firstName>Peter</firstName>
        <lastName>Pan</lastName>
        <email>secondStarLeft@hotmail.com</email>
        <phone>4175554321</phone>
      </contact>
      <contact id="6" type="business">
        <firstName>ACME Code Works</firstName>
        <lastName/>
        <email>acmecodeworks@gmail.com</email>
        <phone>3145554132</phone>
      </contact>
    </contacts>
  </fx:XML>
</fx:Declarations>
```

NOTE

Valid XML must always contain a root tag. In this example, you must wrap the list of contacts in the root tag `<contacts>` `</contacts>`. You can call this tag anything you want, because it isn't going to be referenced anywhere. In the application code, we'll reference the entire XML script using its MXML object ID—`contactsXML`.

You might recognize the `xmlns=""` attribute. It's a namespace property, and after all, they've been near the top of every Flex example you've created. When creating the `<fx:XML/>` tag in Source mode, code completion will insert the blank namespace for you. The namespace is not necessary for simple XML, but its presence won't interfere with compilation or introduce compile problems.

Handling XML Data as an `XMLListCollection`

Now that we've prepared a block of XML data, we need somewhere to put it. To start, let's just dump the `firstName` values into a `List` control; to do so, place the code in Example 11-12 directly below the Declarations section in your application.

Example 11-12. Assigning XML data as the source of an `XMLListCollection` `dataProvider` object

```
<s:List id="list">
  <s:dataProvider>
    <s:XMLListCollection source="{contactsXML.contact.firstName}"/>
  </s:dataProvider>
</s:List>
```

Note the required format of the `dataProvider`: `XMLListCollection`. Spark `List` controls require XML data to be strongly typed `XMLListCollection` objects. Alternatively, you can get the same result from a Halo `List` without converting it as an `XMLListCollection`, as in Example 11-13.

Example 11-13. Assigning XML data directly to a Halo `List` control

```
<mx:List id="list" dataProvider="{contactsXML.contact.firstName}"/>
```

It's certainly shorter and sweeter. Plus, it works. But the Halo implementation is possible only because the original XML data object is coerced behind the scenes into an `XMLListCollection` object for use in the `List` control.

A few conditions make the Halo usage dodgy. Because XML can originate from any number of sources—third-party websites, databases, or external XML files—the basic XML object was not designed to *listen* for changes (such as the sort that can occur within your application) to its underlying data. As such, the preferred strategy is to cast XML data as an `XMLListCollection`, which is a more appropriate source for data binding.

NOTE

In this context, the statement `contactsXML.contact.firstName` is parsing XML data using a standard known as E4X, or ECMAScript for XML. We discuss E4X properly in the next section.

NOTE

Although the `<fx:XML>` data object does not listen for changes to its source, XML data objects can be manipulated using *ActionScript*. So don't confuse this as meaning that XML objects cannot be modified with Flex; rather, the point is that XML objects don't listen for changes.

Reading XML Using E4X

Next we investigate how we pulled the **firstName** values out of the XML object. Referring back to the Spark example, the following line of code is responsible for identifying the source values for the **XMLListCollection**:

```
<s:XMLListCollection source="{contactsXML.contact.firstName}"/>
```

It's called *E4X*, or *ECMAScript for XML*, and it hugely simplifies working with XML data in Flex. Using E4X, we can step through the XML nodes using dot notation. Furthermore, E4X gives us a neat card trick. Notice what happens if you replace the node-by-node reference shown earlier with another using two dot operators in succession (**..**), the so-called *descendant accessor*, shown in Example 11-14.

Example 11-14. Quickly accessing an XML node structure using E4X and the descendant accessor

```
<s:XMLListCollection source="{contactsXML..firstName}"/>
```

If you test them, both results are identical. The descendant accessor allows for rapid parsing of children or children within children. Voilà! The **dataProvider** becomes a collection of **firstName** values.

Notice also that we didn't bind the **dataProvider** directly to **contactsXML** like this:

```
<s:XMLListCollection source="{contactsXML}"/>
```

Rather, we handled it like this:

```
<s:XMLListCollection source="{contactsXML.contact.firstName}"/>
```

Or like this:

```
<s:XMLListCollection source="{contactsXML..firstName}"/>
```

In both techniques the root tag of our XML is ignored. They begin by referencing the **id** of the Flex MXML object—not the root tag within the XML source, but the MXML object referencing that source. Next, we use E4X to point to the desired nodes in our source XML, in this case, **firstName**. The resulting **dataProvider** becomes a list of first name values.

If we were to set our source equal to **{contactsXML.contact}**, we would end up with a list of XML syntax for each **<contact/>** tag, as shown in Figure 11-5.

Stated simply, **List** controls want a list of items, and the MXML XML class's **id**, **contactsXML**, replaces the top-level node of the XML. Without pointing to a node with an explicit value, the **List** would think you were binding it to a specific element in the XML structure, which isn't necessarily a value, but instead a complete entity of its own, and it would not display correctly. Rather, you build a list of values using an expression that points to the **<firstName/>** tags in your XML. The result is shown in Figure 11-6.

```
<contact id="1" type="person">
  <firstName>John</firstName>
  <lastName>Jingleheimerschmidt</lastName>
  <email>jakejingleheimerschmidt@gmail.com</email>
  <phone>4795552134</phone>
</contact>
<contact id="2" type="person">
  <firstName>Goldie</firstName>
  <lastName>Locks</lastName>
  <email>goldielocks.2010@hotmail.com</email>
  <phone>7015551234</phone>
</contact>
<contact id="3" type="business">
  <firstName>Charlestown Chiefs</firstName>
  <lastName>
  <email>chiefs.hockey@yahoo.com</email>
  <phone>8145554421</phone>
</contact>
<contact id="4" type="person">
  <firstName>Red</firstName>
  <lastName>Ridinghood</lastName>
  <email>redridinghood.2010@gmail.com</email>
  <phone>4155552341</phone>
</contact>
<contact id="5" type="person">
  <firstName>Peter</firstName>
  <lastName>Pan</lastName>
  <email>secondStarLeft@hotmail.com</email>
  <phone>4175554321</phone>
</contact>
<contact id="6" type="business">
  <firstName>ACME Code Works</firstName>
  <lastName>
  <email>acmecodeworks@gmail.com</email>
  <phone>3145554132</phone>
</contact>
```

Figure 11-5. A list of contact nodes in XML syntax

```
John
Goldie
Charlestown Chiefs
Red
Peter
ACME Code Works
```

Figure 11-6. A list of first name values displayed in a List control

The next example revisits the Halo namespace and the **DataGrid** component and shows you another option available using E4X: the appropriately named attribute identifier (@). We'll continue to work with **contactsXML**, so be sure to retain it in the Declarations section of your application.

Handling XML Data in the Halo DataGrid

We introduced the Halo **DataGrid** previously when we discussed populating **List** controls with **Array** data, and you might remember that an **ArrayCollection** is the required **dataProvider** object for Spark **List** controls.

Populating a **DataGrid** with XML data is quite similar. To maintain the Spark approach—and as we saw in previous XML examples—we'll bind our **dataProvider** to an **XMLListCollection** object.

The next example continues using **contactsXML**. Assuming the XML object remains in your Declarations section, create a **DataGrid** as in Example 11-15.

Example 11-15. Assigning data to a Halo DataGrid using a nested dataProvider

```
<mx:DataGrid horizontalCenter="0" verticalCenter="0">
  <mx:columns>
    <mx:DataGridColumn width="35" headerText="ID" dataField="@id"/>
    <mx:DataGridColumn width="70" headerText="Type" dataField="@type"/>
    <mx:DataGridColumn width="150" headerText="First" dataField="firstName"/>
    <mx:DataGridColumn width="250" headerText="Email" dataField="email"/>
  </mx:columns>
  <mx:dataProvider>
    <s:XMLListCollection source="{contactsXML.contact}"/>
  </mx:dataProvider>
</mx:DataGrid>
```

For the most part, we have the same basic **DataGrid** construction—a **DataGrid** complete with **columns** and **DataGridColumn** containers. Following the closing tag of the **columns** block, we have a **dataProvider** as an **XMLListCollection** pointing to the **<contact/>** nodes in the XML source. In the meantime, each **DataGridColumn** object points to a specific property or field in the **<contact/>** node.

We used E4X dot notation to point our **dataProvider** at the list of **<contact/>** objects. And as you likely remember, the **dataField** property of each **DataGridColumn** specifies which element of the data set that column will handle.

However, this approach does something new. Notice the @ operator in these expressions: **dataField="@id"** and **dataField="@type"**. By merely looking at the structure of **contactsXML**, you probably realize what's happening. The *attribute operator* (@) preceding the **id** and **type** attribute names tells our **DataGridColumn** objects that their display values are tag attributes of each **<contact/>** node in the **dataProvider**.

NOTE

Why the @ sign? Well, say it out loud: “the ‘at’ sign.” We’re dealing with “at”-tributes, so it made sense to use that symbol.

The attribute operator is merely a convenient way to show that you want to access a tag attribute. Because XML can have both attributes and child/content tags, you need a special approach to distinguish between them. Figure 11-7 shows the results of the **DataGrid** absorbing our **XMLListCollection**.

ID	Type	First	Email
1	person	John	jakejingleheimerschmidt@gmail.com
2	person	Goldie	goldielocks.2010@hotmail.com
3	business	Charlestown Chiefs	chiefs.hockey@yahoo.com
4	person	Red	redridinghood.2010@gmail.com
5	person	Peter	secondStarLeft@hotmail.com
6	business	ACME Code Works	acmecodeworks@gmail.com

Figure 11-7. The **DataGrid** populated with XML contact info

Alternatively, if you wanted to populate the **DataGrid** without taking the Spark approach, all you'd have to do is declare the **dataProvider** using inline **ActionScript**, pointing it directly at the **<contact/>** node in **contactXML**. As Example 11-16 demonstrates, everything else remains the same.

Example 11-16. Assigning data directly to a Halo **DataGrid** by binding to its **dataProvider** property

```
<mx:DataGrid dataProvider="{contactsXML.contact}">
  <mx:columns>
    <mx:DataGridColumn width="35" headerText="ID" dataField="@id"/>
    <mx:DataGridColumn width="70" headerText="Type" dataField="@type"/>
    <mx:DataGridColumn width="150" headerText="First" dataField="firstName"/>
    <mx:DataGridColumn width="250" headerText="Email" dataField="email"/>
  </mx:columns>
</mx:DataGrid>
```

Loading External Data at Compile Time

Any long-winded XML data set will become an irritation if it's built into the body of your code. You'll likely find yourself thinking the source is getting out of hand. Just imagine the difficulties working with the code editor if the contacts included hundreds of entries.

Fortunately, it's easy to work with an external XML file and load it into the application when it's compiled. The **<fx:XML/>** tag has a **source** property that you can point to outside data, and here we demonstrate how to benefit from it by migrating the content of **contactsXML** into an external file.

Begin by creating a new folder under your **src** directory. In the Package Explorer, make sure your project is expanded, then select the **src** folder, and open **File→New→Folder**. When the dialog appears, name the folder **xml**. Clicking **Finish** should create a new folder called **xml** in your **src** path. The icon representing the **xml** folder may appear somewhat unusual, but if it looks like the one in Figure 11-8, you're on the right track.

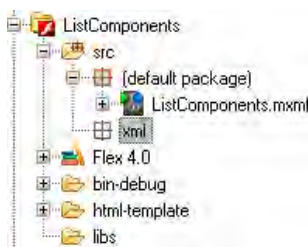


Figure 11-8. A new **xml** folder in the **src** directory for a project

Next we'll create the XML file. With the fresh *xml* folder selected, open File→New→File. When the dialog appears, provide the filename *contacts.xml*. Be sure to include the *.xml* extension. Once you hit Enter or click Finish, the XML file will be created for you and a new editor pane will be opened.

On line 1 of your new XML file, add the following header:

```
<?xml version="1.0" ?>
```

Complete the file by creating the content for **contactsXML**. You might as well copy and paste everything between the **<fx:XML>** opening tag and the **</fx:XML>** closing tag from the **contactsXML** declaration in your project. Make sure to leave the MXML tags for **contactsXML**, because we're not quite finished with them. Once the content is moved, save the file (File→Save).

This completes the external XML file. You can always test an XML file by pointing your web browser at it. If the XML syntax is arranged correctly, the browser should parse it and produce something similar to Figure 11-9. On the other hand, if the browser can't parse it, take a close look at the XML, because you likely have a syntax issue.

NOTE

If you're working with an external XML file and you can't identify the source of errors, try loading the XML file in a web browser to see if it displays at all or as you believe it should. This is a good way to rapidly test XML.

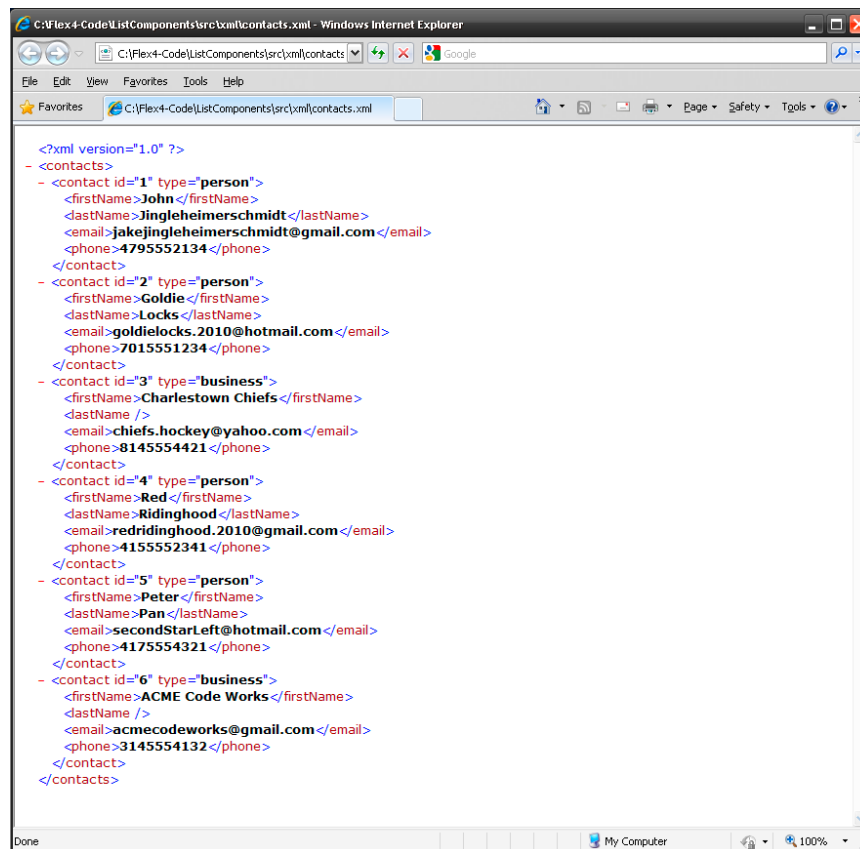


Figure 11-9. Web browser output for a valid XML file

Now the XML data is conveniently located outside the main source code. But you still need to pull it into your project. You should still have the **contactsXML** tag in your **Declarations** section, so rework it to include the change emphasized in Example 11-17. Either your Spark implementation or your Halo **DataGrid** will work fine for the rest of the example.

Example 11-17. Filling an XML data object with XML from an external asset

```
<fx:Declarations>
  <fx:XML id="contactsXML" source="/xml/contacts.xml"/>
</fx:Declarations>

<mx:DataGrid horizontalCenter="0" verticalCenter="0">
  <mx:columns>
    <mx:DataGridColumn width="35" headerText="ID" dataField="@id"/>
    <mx:DataGridColumn width="70" headerText="Type" dataField="@type"/>
    <mx:DataGridColumn width="150" headerText="First" dataField="firstName"/>
    <mx:DataGridColumn width="250" headerText="Email" dataField="email"/>
  </mx:columns>
  <mx:dataProvider>
    <s:XMLListCollection source="{contactsXML.contact}"/>
  </mx:dataProvider>
</mx:DataGrid>
```

At this point, you can go ahead and run it.

There is one drawback to loading external data at compile time. Specifically, the contents of our XML file get compiled into our Flex application, making it heavier and “more expensive” to load. For this reason, this approach is called loading external data at *compile* time. If you want to avoid compiling your XML data into your project, though, another option exists, which we discuss next.

Loading External Data at Runtime

Sometimes it’s desirable to pull in external data at runtime, which means our application will be fully loaded by the web browser, and then the **application-Complete** event, or some other event, will trigger a request for external data. In our case we’ll request *contacts.xml*. To carry out this procedure, we’ll take advantage of the **HTTPService** component to handle request and result operations.

One of the greatest advantages of this approach is the liberty to change the contents of your source XML file at your leisure, without having to recompile your project.

The **HTTPService** component provides a **send()** method to get text or XML content over the Web using standard HTTP. This means you can load information from websites in HTML, or even sites created in PHP, ASP, or other server technologies that generate HTML content.

WARNING

Compiling data into your project (i.e., loading data at compile time) results in larger SWFs, which may take longer to download. On the upside, you know the data will be available to your application.

HTTP

HTTP stands for Hypertext Transfer Protocol, and it’s a standard for transferring information across the Internet. This protocol is used when opening a web page in a browser; it sends all the text and images you see displayed in your browser.

When you type an address in your browser to go to a web page, you generally add **http://** at the beginning of the address. The **http://** part simply means “use the HTTP protocol.”

To use an **HTTPService** component, we'll place it in the Declaration section of our application, assign it an **id** property, and point it toward a valid URL.

You have the option to specify either an absolute or a relative URL. An *absolute URL* looks like <http://www.learningflex4.com/xml/contacts.xml>. However, a *relative URL* is a resource contained within or above the root directory of a website; for example, [xml/contacts.xml](#) would be a relative URL. Because we'll be using a local file, we'll aim the **HTTPService** component using a relative URL.

Into the Bin

Running your application in Flash Builder for a project deployed to the Web will launch an HTML file containing a compiled version of the application. More happens at compile time than the mere creation of a SWF. For one, any files you placed in your source folder are copied to the **bin-debug** folder. If you ran the example demonstrating how to load data at compile time, you can open your **bin-debug** folder and find both the **xml** folder and the file **contacts.xml**.

In other words, when you run your application, the copy of the compiled file in your **bin-debug** folder is what you'll see. This applies to more than text files; any media you add to your application's **src** folder will be copied into **bin-debug**.

Knowing this, be aware that any edits you make to files in the **bin-debug** folder will be summarily overwritten, so be sure to modify only the original version, located in the **src** folder.

Because this example requires code in every section of the application (e.g., Script/CDATA, **Declarations**, structural XML, etc.) we'll create a fresh application in your current project directory. Expand the **src** folder (if you haven't already), select the default package, and follow File→New→MXML Application. When the dialog appears, name the project **ExternalXML** and select Finish.

Start by declaring the **HTTPService** component within the **Declarations** section, as in Example 11-18.

Example 11-18. Creating an **HTTPService** component to load the external XML file

```
<fx:Declarations>
  <s:HTTPService id="contactsHTTP" url="xml/contacts.xml"
    resultFormat="e4x"/>
</fx:Declarations>
```

This creates an **HTTPService** component, gives it a valid **id**, points it to the **contacts.xml** file you created earlier, and lets the service know you'll be loading XML data. Notice how we set up the relative URL: **url="xml/contacts.xml"**.

NOTE

The values available to the **resultFormat** property include **array**, **e4x**, **flashvars**, **object**, **text**, and **xml**.

If you don't set the **resultFormat** property, by default the **HTTPService** component will convert any data it receives into regular objects. Thus, to tell the component that you're loading XML data and that you want to use E4X operations, set the **resultFormat** property to **e4x**.

Now build a trigger to call the **HTTPService** component's **send()** method. For this, take advantage of the **applicationComplete** event, which fires once the SWF is fully drawn. To do this, append your application's opening tag attributes to handle a response to its **applicationComplete** event, as shown in Example 11-19.

Example 11-19. Using the *applicationComplete* event to call the *HTTPService* component's *send()* method

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  applicationComplete="onAppComplete">
```

Assigning the event attribute tells the application, "Once you're fully built, call **onAppComplete()**." Now we need to build that function. So start a **Script/CDATA** block in the appropriate area of your application. Code completion should help you. Inside the block, add the function shown in Example 11-20.

Example 11-20. Calling the *HTTPService* component's *send()* method

```
<fx:Script>
  <![CDATA[
    private function onAppComplete():void{
      contactsHTTP.send();
    }
  ]]>
</fx:Script>
```

Once the compiled application is drawn, it calls **onAppComplete()**, which uses **contactsHTTP** to request the resource at *xml/contacts.xml*.

The final step is creating the **DataGrid** and assigning its **dataProvider** object. You know **contactsHTTP.lastResult** will be XML-formatted data, but you need to bind the **dataProvider** to the list of contacts. So, the **XMLListCollection** **source** property should be **contactsHTTP.lastResult.contact**. This binds the **dataProvider** to the list of contact nodes in the most recent invocation of the service component's **send()** request. Compared to previous examples of the **DataGrid**, only the **dataProvider** component's **source** property is changed in Example 11-21.

Example 11-21. Binding the `DataGrid` control's `dataProvider` object to the `lastResult` property of the `HTTPService` component

```
<mx:DataGrid horizontalCenter="0" verticalCenter="0">
  <mx:columns>
    <mx:DataGridColumn width="35" headerText="ID" dataField="@id"/>
    <mx:DataGridColumn width="70" headerText="Type" dataField="@type"/>
    <mx:DataGridColumn width="150" headerText="First" dataField="firstName"/>
    <mx:DataGridColumn width="250" headerText="Email" dataField="email"/>
  </mx:columns>
  <mx:dataProvider>
    <s:XMLListCollection source="{contactsHTTP.lastResult.contact}"/>
  </mx:dataProvider>
</mx:DataGrid>
```

Now the service will be invoked when the application is opened, the data will be pulled in, and the binding will connect the XML data to your `DataGrid`.

It's Not Your Fault

When using an `HTTPService`, the data returned is reliant upon the remote host. Just like a web browser, it's using HTTP protocol. Have you ever tried to go to a website and the page wouldn't load? This can happen with the `HTTPService` component as well. If your `send()` request is compromised, you may get a fault. This means there's a problem somewhere; perhaps the file you're requesting no longer exists, or the web server is offline.

If the server fails to respond, the `lastResult` property will not be updated as expected. By default, Flex will pop up an alert informing you of the problem. If you'd like a custom response, though, you can listen for loading errors with the `HTTPService fault` event:

```
<s:HTTPService id="contactsHTTP" url="xml/contacts.xml"
  resultFormat="e4x" fault="faultHandler(event)"/>
```

This event fires whenever a fault occurs, and in this case, it would call an `ActionScript` function, `faultHandler`, and pass it the `event` object.

Of course, you can take this skill of connecting to remote data and apply it to data elsewhere on the Web. For a quick example, instead of loading `contacts.xml` using a relative local path, use an XML file residing on a remote web server. To do so, simply change the `url` property of the service to point to the remote file, which is available through the companion website:

```
<s:HTTPService
  id="contactsHTTP" resultFormat="e4x"
  url="http://learningflex4.com/xml/contacts.xml"/>
```

If you haven't already, go ahead and run the application to appreciate your work. Now the application loads an XML file, not from your local disk, but across the Web. Because the file is traveling on the Internet, you might notice

NOTE

Remote data is not accessible to a Flex application unless the remote site has enabled access via a special policy file. See the sidebar "Playing Nice in the Sandbox" on page 228 for more information.

NOTE

For an absolute URL, you must use the `http://` protocol designation.

a bit of latency. By default, Flex service components will display a pinwheel-style status indicator, or “busy cursor,” to let you know that data is being downloaded. If you want to remove this, set the service’s `showBusyCursor` to `false`, like this:

```
<s:HTTPService
    id="contactsHTTP" resultFormat="e4x" showBusyCursor="false"
    url="http://learningflex4.com/xml/contacts.xml"/>
```

Playing Nice in the Sandbox

When talking about Flash, or computer security in general, you’re going to hear the term *sandbox* a lot. A sandbox is a secure location for a program (such as Flash Player) to run and do what it pleases. It is a restrictive space that prevents the program from breaking away and to which other programs may have limited access. Think of a parent taking their kids to the park to play in an actual sandbox; it’s a place where the child is free to play without the parents worrying about their safety.

Concerning Flash Player, a very secure sandbox prevents Flash web applications from accessing a user’s filesystem or gathering and manipulating data from a web host operating from an external domain, which means any domain different from the SWF’s parent domain. You’re likely to encounter the most prevalent sandbox restrictions when accessing remote data.

Simply put, a Flex web application isn’t allowed to access data outside the domain to which the application is deployed—unless the owner of that website specifically allows it. If allowed, permission must be granted via an XML file called *crossdomain.xml*, placed on the root of the website.

The *crossdomain.xml* file lets the owner explicitly allow certain domains to access his data. And if he so chooses, he can allow everyone access to his data. To grant your application access to the *contacts.xml* file on the companion website, we created a cross-domain file to allow access to everyone. You can view this cross-domain file at <http://www.learningflex4.com/xml/crossdomain.xml>. It looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
    SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all"/>
    <allow-access-from domain="*" />
    <allow-http-request-headers-from domain="*" headers="*" />
</cross-domain-policy>
```

If the cross-domain file wasn’t there or didn’t allow everyone access, downloading the *contacts.xml* file into your Flex application would result in an error, and you wouldn’t get the data.

In the next section, we demonstrate how to listen for and react to item selection in the **List**-based components.

Implementing List Selection

A **List** or List-based control not only shows a list of items but listens for item selection as well. If any item/row in your list is selected, the entire data range for that row will become available through the **List** control's **selectedItem** property. For the following example, either convert your **ExternalXML** application or create a new application called **ListSelApp**.

In the demo in Example 11-22, the **Panel** container's labels will display the values corresponding to the selected item/row in the **DataGrid**. In addition to the purpose of the exercise, which is demonstrating **List** selection, notice how the **Panel** container is taking advantage of layout bindings to establish both its position and size.

Example 11-22. Source code for the *ListSelApp* application

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  applicationComplete="onAppComplete()">

  <fx:Script>
    <![CDATA[
      private function onAppComplete():void{
        contactsHTTP.send();
      }
    ]]>
  </fx:Script>

  <fx:Declarations>
    <s:HTTPService
      id="contactsHTTP" resultFormat="e4x" url="xml/contacts.xml"/>
  </fx:Declarations>

  <s:layout><s:BasicLayout/></s:layout>

  <s:VGroup id="vgroup" horizontalCenter="0" top="50">
    <mx:DataGrid id="contactsDataGrid">
      <mx:columns>
        <mx:DataGridColumn
          width="35" headerText="ID" dataField="@id"/>
        <mx:DataGridColumn
          width="70" headerText="Type" dataField="@type"/>
        <mx:DataGridColumn
          width="150" headerText="First" dataField="firstName"/>
        <mx:DataGridColumn
          width="250" headerText="Email" dataField="email"/>
      </mx:columns>
      <mx:dataProvider>
        <s:XMLListCollection source="{contactsHTTP.lastResult.contact}"/>
      </mx:dataProvider>
    </mx:DataGrid>
  </s:VGroup>
```

WARNING

If you create a new application for this exercise, make sure to maintain a link to the contacts.xml file.

NOTE

You might be curious about this use of a **FormItem** container outside a **Form**. The **FormItem** container provides a convenient mechanism of adding labels to UI components, although you could manually create it by placing a **Label** control inside an **HGroup**.

```
<s:Panel title="Info for Selected Contact"
  y="{vgroup.y+vgroup.height}" x="{vgroup.x}" width="{vgroup.width}">
  <s:layout>
    <s:VerticalLayout
      paddingTop="5" paddingBottom="5"
      paddingLeft="5" paddingRight="5">
    </s:VerticalLayout>
  </s:layout>
  <mx:FormItem label="First Name:">
    <s:Label text="{contactsDataGrid.selectedItem.firstName}"/>
  </mx:FormItem>
  <mx:FormItem label="Last Name:">
    <s:Label text="{contactsDataGrid.selectedItem.lastName}"/>
  </mx:FormItem>
  <mx:FormItem label="Email:">
    <s:Label text="{contactsDataGrid.selectedItem.email}"/>
  </mx:FormItem>
  <mx:FormItem label="Phone:">
    <s:Label text="{contactsDataGrid.selectedItem.phone}"/>
  </mx:FormItem>
</s:Panel>

</s:Application>
```

In this example, the **phone** value is present in the **Panel** but omitted in the **DataGrid**. Even though the **phone** value was not referenced by the **DataGrid**, that value remains available through the **dataProvider**. Said differently, a List-based control's **selectedItem** property can reference any **dataField** in its underlying **dataProvider** object, including fields not consumed by the **DataGrid** itself.

Once you're ready, of course, compile and run the application. The result is shown in Figure 11-10.

NOTE

The **List** controls have a property called **allowMultipleSelection** that, when set to **true**, allows you to select more than one item at a time. To access the selected items, you use the **selectedItems** property instead, which produces an **Array** of items.

ID	Type	First	Email
1	person	John	jakejingleheimerschmidt@gmail.com
2	person	Goldie	goldielocks.2010@hotmail.com
3	business	Charlestown Chiefs	chiefs.hockey@yahoo.com
4	person	Red	redridinghood.2010@gmail.com
5	person	Peter	secondStarLeft@hotmail.com
6	business	ACME Code Works	acmecodeworks@gmail.com

Info for Selected Contact	
First Name:	John
Last Name:	Jingleheimerschmidt
Email:	jakejingleheimerschmidt@gmail.com
Phone:	4795552134

Figure 11-10. Data bindings trigger changes to the label text corresponding to the currently selected item in the **DataGrid** component

Connecting to Search Results

In this section we consider an example that demonstrates working with a third-party library as well as how to connect to dynamic search results. To do this, we'll start a new Flex project called **YahooSearch**.

Google offers a public API that grants access to Flash and Flex applications, but it requires the developer to understand and parse the XML data it returns. Alaric built an easy-to-use Yahoo! search component that makes it simple to place Yahoo! search data in a Flex application.

To follow along with the example, you'll need the ActionScript 3 Search API. To get it, visit <http://developer.yahoo.com/flash/astra-webapis/>, and download the Yahoo! ASTRA Web APIs library. You'll get a ZIP file that, once extracted, will have a few folders standard to most Flex or ActionScript libraries: *Build*, *Documentation*, *Examples*, and *Source*. It may interest you that the source code is available in the *Source* folder, but for the example, we're interested in the *Build* folder and a SWC file named *AstraWebAPIs.swc*.

Once you identify the SWC, simply copy and paste, or even drag and drop it, into the *libs* folder of your **YahooSearch** project. Once you do this, the **SearchService** component will be available to your application.

So how do you use this component? First, let's recap the concept of namespaces.

As you should recall, a namespace, in terms of Flex, is a way to distinguish groups of related components. You'll be using a component called **SearchService**, but this component isn't part of the default Flex framework, so it's not going to use the native Flex namespaces: **fx**, **s**, or **mx**. Instead, it will use a namespace called **yahoo**. However, as you've learned, typing the namespace in Flash Builder's Source mode isn't necessary when utilizing code completion, so when you want to add the **SearchService** component, you can start typing **<SearchService** and, if you allow code completion to complete your tag, the namespace will be inserted automatically (Figure 11-11). If not, you can deduce how to add the namespace manually in the next example.

The **SearchService** component needs just one property to make it work: a search string, which is defined by its **query** attribute. In our example, we'll bind the **query** property to a **TextInput** (**queryTI**), and we'll use a **Button** to call the **SearchService** component's **send()** method.

SWC Files

A **SWC** file (usually pronounced "swik") is a special library file loaded with precompiled controls, components, and methods. SWC libraries are convenient for three reasons:

- A SWC is a single file and easier to distribute than source code.
- If you don't want to distribute your source code, a SWC gives you a certain degree of protection because unraveling the source code requires a decompilation utility.
- Because SWC files are precompiled, your application does not have to compile them every time you rebuild.

In fact, the Flex framework you've been using is contained in a few separate SWC files.

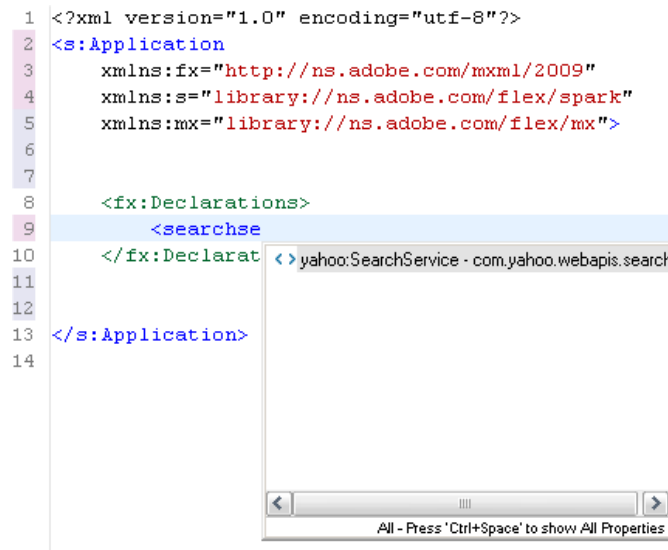


Figure 11-11. Finding the `yahoo:SearchService` component using code completion

Although the `SearchService` component isn't related to the `HTTPService` component, it does have similar internals making it work. It uses a method called `send()` and a property called `lastResult`, which makes it easy for Flex developers to get started.

You can build the `YahooSearch` application with the code in Example 11-23; lines of note are emphasized to help focus your attention.

Example 11-23. *The YahooSearch application*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:yahoo="http://www.yahoo.com/astra/2006/mxml"
  defaultButton="{searchButton}">

  <fx:Declarations>
    <yahoo:SearchService id="searchService"
      applicationId="YahooSearch"
      query="{queryTI.text}"/>
  </fx:Declarations>

  <s:VGroup left="10" right="10" top="10" bottom="10">
    <s:Label text="Yahoo! Search:" fontWeight="bold"/>

    <s:HGroup>
      <mx:FormItem label="Query:" fontWeight="bold">
        <s:TextInput id="queryTI" width="350"/>
      </mx:FormItem>
    </s:HGroup>
  </s:VGroup>
</s:Application>
```

```

        <mx:FormItem>
            <s:Button id="searchButton" label="Search"
                click="searchService.send()"/>
        </mx:FormItem>
    </s:HGroup>

    <s:List id="resultsList" width="100%" height="100%">
        <s:dataProvider>
            <s:ArrayCollection source="{searchService.lastResult}"/>
        </s:dataProvider>
    </s:List>

</s:VGroup>
</s:Application>

```

Besides the `<yahoo:SearchService/>` component, which is declared in the Declarations section, the `defaultButton` attribute of the `Application` container is also new. We address that in a moment.

Here's a question. How did we know to absorb the `lastResult` method into an `ArrayCollection`? Well, when you're assigning attribute values to the `SearchService` component, or later when you're referencing the `lastResult` method with inline ActionScript, just start typing `lastRe`—like you're referencing the `lastResult` property—and code completion will reveal that the returning data will be an `Object` (Figure 11-12).



Figure 11-12. Discovering the data type returned from the `SearchService` component's `lastResult` method

As for the `defaultButton` property, this property binds a specified `Button` control to the Enter/Return key so that pressing the Enter/Return key fires the button's `click` event, in this case submitting the search. It also provides a visual cue by highlighting the `Button`.

Go ahead and run the application. Enter a search string, and then click the `Button` (or press the Enter/Return key) to submit your query. Assuming everything works as expected, data will return from Yahoo!, and some results will appear in the `List` control. By default, the `List` displays the title of the page, and it could use some help. We'll learn how to improve on that in a moment. You should see something similar to Figure 11-13.

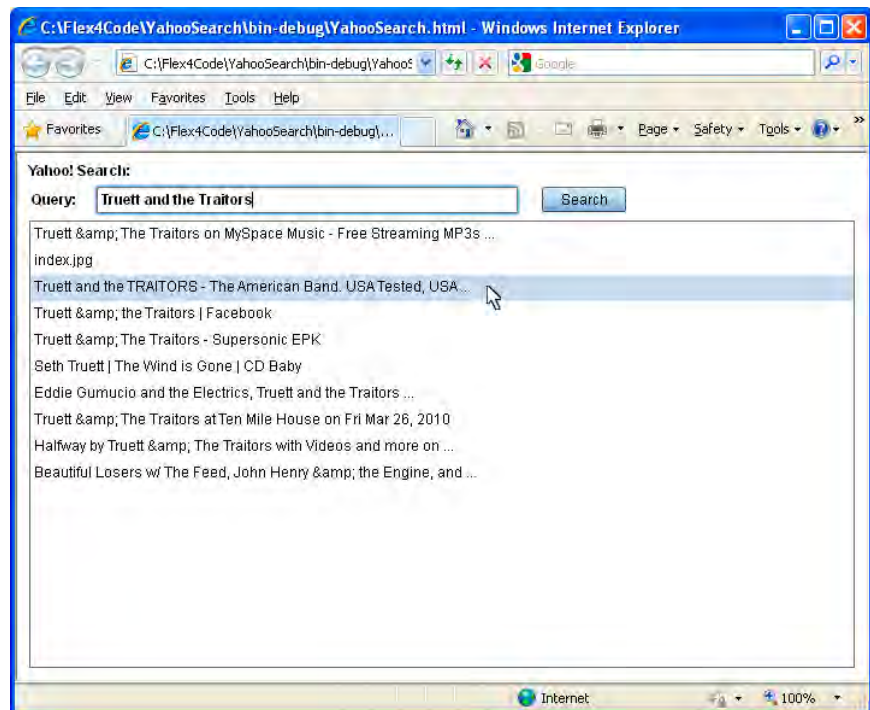


Figure 11-13. A simple, dynamic search application using a third-party API

Next, you'll learn how to expand the potential of the search tool by adding drag-and-drop functionality.

Dragging and Dropping in Lists

Built into Flex's **List** controls is the ability to easily drag and drop data. No complex code is necessary to drag an item from one **List** and drop it into another. To toggle the feature, simply enable two properties: **dragEnabled** and **dropEnabled**.

For example, add another **List** control to the search application, and give it an empty **ArrayCollection** **dataProvider**. Then, set the new **List** component's **dropEnabled** property to **true**. The **dropEnabled** property allows a **List** to accept data items that are dropped into it.

With the new **List** component created, return to **resultsList**, declare the **dragEnabled** property, and set it to **true**. The **dragEnabled** property tells **resultsList** to allow users to drag its items. Your changes should resemble the code in Example 11-24.

Example 11-24. Adding drag-and-drop support to two *List* controls

```

<s:List id="resultsList" width="100%" height="100%" dragEnabled="true">
  <s:dataProvider>
    <s:ArrayCollection source="{searchService.lastResult}"/>
  </s:dataProvider>
</s:List>

<s:List id="dropList" width="100%" height="100%" dropEnabled="true">
  <s:dataProvider>
    <s:ArrayCollection/>
  </s:dataProvider>
</s:List>

```

Now when you run the application, you can simply drag a favorite search result from the **resultsList** and drop it into the second **List**. When you perform a drag-and-drop between **List** controls, the entire item (including its properties) is copied from the **dataProvider** of the first **List** into the **dataProvider** of the second **List** (Figure 11-14).

Dragging and dropping works with other **List** controls as well, such as **TileList** and **DataGrid**. Here's something to keep you thinking: if you replaced **dropList** with a **DataGrid**, each time you dragged an item into the **DataGrid**, you could see *every* property of the search result item displayed as a column.

NOTE

By default, dragging and dropping from one list to another copies the data. There is also a property called **dragMoveEnabled**, which lets you move (i.e., cut) items from one **List** control to another.

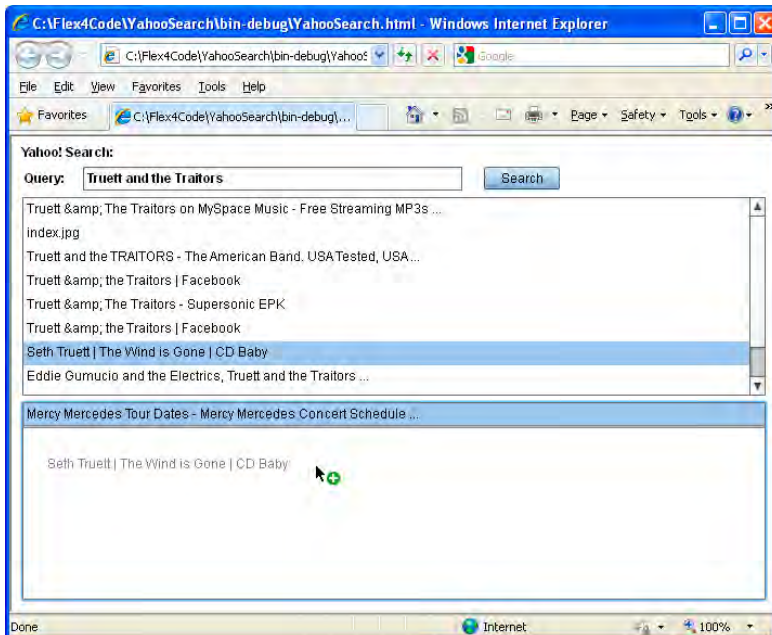


Figure 11-14. Dragging and dropping list items in the YahooSearch application

Creating Custom Item Renderers

Here's another trick: you can customize the row display characteristics for List-based controls by developing custom **ItemRenderer** classes. Once you develop an **ItemRenderer**, you can use it to override a List-based control's default labeling mechanism. Furthermore, code completion makes it fairly easy to create a custom **ItemRenderer**. Let's see if you agree.

To get things underway, either delete or comment out the second **List** from the previous example, and then delete the **dragEnabled** property from the **resultsList** attributes. Next, prepare **resultsList** by giving it a nested **VerticalLayout**, as shown in Example 11-25.

Example 11-25. *Preparing a List to receive a custom ItemRenderer*

```
<s:List id="resultsList" width="100%" height="100%">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <s:dataProvider>
    <s:ArrayCollection source="{searchService.lastResult}"/>
  </s:dataProvider>
</s:List>
```

Now let's bring in the external source directory we've been building throughout the text. We've been storing our external source in the path `C:\Flex4Code\com`; so add that directory, or your equivalent, as a source class-path (Project→Properties→Build Path→Source path). Once the external source is in place, make sure it's selected in the Package Explorer, and then create a new package (File→New→Package) named *learningflex4.renderers*. This is where we'll save our custom **ItemRenderer**. Now we're ready to create the **ItemRenderer**.

After the **List** component's size properties, provide a space and start typing **itemRend**—code completion should recommend the **itemRenderer** property. Go ahead and select it, and if you are not subsequently presented with the option "Create Item Renderer...", hit Ctrl-space bar to toggle auto-completion. The auto-completion option should resemble Figure 11-15.

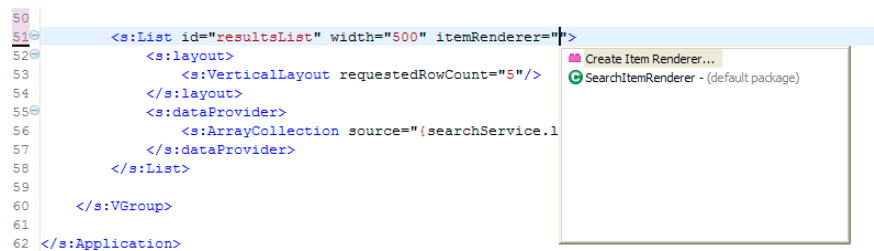


Figure 11-15. Selecting "Create Item Renderer..." from the code completion menu

Selecting “Create Item Renderer...” will present you with a dialog to configure your new **ItemRenderer**. First, set the *Source* folder value by browsing to *[source path] com*, which should be under the **YahooSearch** project. Then, browse to the package you just created, *learningflex4.renderers*. Finally, name the class **SearchItemRenderer** and hit Enter/Return. You’ll emerge into a fresh code editor for your new **ItemRenderer** class. The default code should resemble Example 11-26.

Example 11-26. *A brand-new ItemRenderer*

```
<?xml version="1.0" encoding="utf-8"?>
<s:ItemRenderer
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    autoDrawBackground="true">

    <s:Label text="{data}"/>

</s:ItemRenderer>
```

NOTE

Before you get deep into this exercise, you may want to confirm that your **List** gained the following property defining its **itemRenderer**:

```
<s>List id="resultsList" width="100%" height="100%"
    itemRenderer="learningflex4.renderers.SearchItemRenderer">
```

Within the new code editor, we’ll define a composite component to replace the default **Label** of the **List** control. First delete the existing instance of the Spark **Label** component, and then create the component code shown in Example 11-27 in your editor. As usual, we’ll emphasize the significant lines and discuss them momentarily.

Example 11-27. *Code for the Custom ItemRenderer component*

```
<?xml version="1.0" encoding="utf-8"?>
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    autoDrawBackground="true" width="100%">

    <fx:Script>
        <![CDATA[
            import flash.net.navigateToURL;
            import flashx.textLayout.elements.TextFlow;
            import spark.utils.TextFlowUtil;

            // merge name and summary and return as TextFlow
            private function renderText(name:String, summary:String):TextFlow{
                var textFlow:TextFlow =
                    TextFlowUtil.importFromString(name + summary);

                return textFlow
            }
        ]]>
```

NOTE

We only emphasized the renderer’s **width** property near the top, because it seemed like it might go unnoticed.

```

        private function onClick(event:MouseEvent, url:String):void{
            var urlRequest:URLRequest = new URLRequest(url);
            navigateToURL(urlRequest, "_blank");
        }
    ]]>
</fx:Script>

<s:VGroup width="100%" gap="2">
    <s:Label text="From: {data.url}" fontWeight="bold"
        buttonMode="true" click="onClick(event, data.url)"/>

    <s:TextArea width="100%" heightInLines="1" editable="false"
        borderVisible="false" verticalScrollPolicy="off"
        textFlow="{renderText(data.name, data.summary)}/>
</s:VGroup>

</s:ItemRenderer>

```

NOTE

For **ItemRenderer** classes, **{data}** represents a row's individual data properties for any given **dataProvider**.

The purpose of this component is to provide a more complete and aesthetic search result in our **List**. We accomplish this by grouping a **Label** and a **TextArea** control together and arranging them to display the search results' **url**, **name**, and **summary** properties. In spite of its size, this component packs quite a functional punch.

To start, we have two **ActionScript** functions doing things we've never seen before—particularly, harnessing the **TextFlow** utilities and calling a URL to open in a new browser window. Then, below in the **MXML**, we have some curious properties. We'll begin our explanation with the **MXML**.

NOTE

Each of these results is actually a class called **WebSearchResult**, which is included as part of the **ASTRA Web APIs** library. It contains properties such as **name**, **summary**, and **clickURL**. If you would like to learn more about this class, consult the documentation that came with the **Yahoo! ASTRA Web APIs** library.

It's important not to lose sight of the focus here; we're creating a composite component capable of rendering relatively complex yet patterned output in a **List** control. The component's inherited **data** property, which references the various item values in the **dataProvider**, is the essential piece to the puzzle. Here we're using **data.url**, **data.name**, and **data.summary** to access those properties in the search results. We'll render the URL in a **Label**, and we'll combine the name and summary properties in a **TextArea** control.

In the **Label** control's declaration, notice how we're using a binding to catch a result's **url** property. This component is also set to recognize **buttonMode**, which will create the familiar pointer-finger on mouseover, and a **click** event that will call the named **urlRequest** function, **onClick()**.

In the **TextArea** control, just about everything is new, but the binding of the **textFlow** property to the **renderText()** function is definitely the most crucial line to observe. Otherwise, we've merely assigned a fixed height correlating to a single line of text (**heightInLines**), we've disabled **editing**, and we've eliminated the container's natural border (**borderVisible**). Setting the **verticalScrollPolicy** to **off** was merely an aesthetic choice; we felt one line of text was enough to review a link, so we didn't see any reason to allow our component to create small vertical scroll bars where descriptions went beyond a single line of text.

NOTE

If you anticipate handling a lot of text markup in your applications, check out the following discussion by Tour de Flex's own Holly Schinsky: <http://devgirl.wordpress.com/2010/04/26/flex-4-and-the-text-layout-framework/>.

Let's return to the `textFlow` property. The `textFlow` property uses inline script to call the `renderText()` method and pass it two strings, `data.name` and `data.summary`. `TextFlow` is a special class introduced with Flex 4 that allows for rendering of complex text objects. In this case, we want to render the character entity markup that's coming through with the search results. In Figure 11-13 you'll see this as `&`; predominantly, which is the HTML entity for the ampersand (&) symbol. In the process, we'll also make our results more unique.

In spite of the component's MXML, the `ActionScript` functions are actually doing all the work. The first function, `renderText()`, takes the search result `name` and `summary`, concatenates them, and then renders them as `TextFlow` using the `TextFlowUtil` class. With this single step accomplished, the function returns the `TextFlow`-formatted product to the `TextArea` control, which uses the formatted text within its `textFlow` property.

The second function handles clicking on the `Label` URL by opening the linked site in a new browser window. This task is accomplished by combining the `URLRequest` class, which must be instantiated and initialized, with the public function `navigateToURL()`, which takes both the instance of the `URLRequest` and a secondary parameter that defines whether to open the link in the same window (`_self`) or another window (`_blank`); if unspecified, opening in a new window is the default.

You should be ready to demo, so compile and run to see what the `YahooSearch` application looks like with a custom `ItemRenderer` (Figure 11-16).

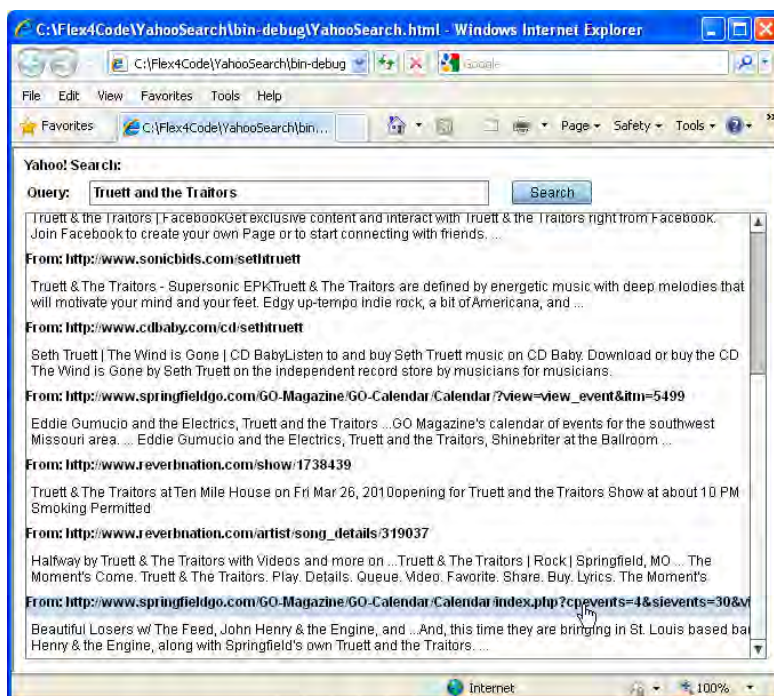


Figure 11-16. The `YahooSearch` application using a custom `ItemRenderer`

NOTE

If your interests require very specific data, you might be surprised by what you find if you do a little digging. For example, the National Oceanic and Atmospheric Administration (NOAA) provides weather information via web services at <http://www.nws.noaa.gov/xml/>, and the USGS recently unveiled a water data web service as an experimental beta at <http://waterservices.usgs.gov/>.

NOTE

WSDL stands for Web Services Description Language, and is a common language model for describing web services. It specifies what operations are available for a service and the format of the service's response.

NOTE

For a directory of web services that you can use, check out <http://www.xmethods.net> or <http://www.webservicelist.com>.

NOTE

Java EE stands for Java Enterprise Edition, and is a version of the Java platform made for server-side processing. ColdFusion is actually a Java EE application, enabling you to create server code in a markup language.

Working with External Data Services

Although the **HTTPService** component is a handy resource, you may require more dynamic alternatives from time to time. In fact, Flex applications thrive when backed by a well-tuned server technology and database combination. For such circumstances, the **WebService** or the **RemoteObject** components, or even PHP, may prove to be the perfect solution for your application.

Web Services

The **HTTPService** component lets you access services that have a URL—particularly data that can be viewed in a browser, such as HTML or XML. However, a number of web services exist that are accessible only through an XML standard called SOAP. Although a bit heavier than other services, SOAP-based services are particularly prevalent in business-to-business applications, and you may find it necessary to connect to a SOAP web service in Flex.

To easily access SOAP web services, use the **WebService** component. You might also find the Connect to Web Service command useful. This command can point to a WSDL URL and automatically generate a good deal of code for you.

To access this command, select Data→Connect to Web Service. From there you'll be prompted with a dialog box that asks you where you'd like to place the code that's generated, and asks for a URL to a WSDL description file. Flash Builder will generate code necessary to access the web service.

RemoteObject

Another powerful data access component is the **RemoteObject**, which lets you easily connect to server-side Java (Java EE or J2EE) or ColdFusion. You gain a number of benefits by using remote objects, which are maps to Java objects and ActionScript objects. This means actual data types such as **Number** or **Date** can be transferred across the wire intact. Another huge benefit of remote objects is that data can be compressed over the wire. That results in speedier access to large amounts of data. Among other benefits, you also can use data push, which allows you to listen for data changes on a remote server and get that new data automatically. Imagine an email application that didn't require you to manually get new mail, but would show you new mail as soon as it arrived.

To use **RemoteObject** component, you must set up your application to use remote object access on a specific web server. Typically this is done when creating a new Flex project.

PHP and HTTPService

PHP remains a popular server solution, and due to the ease of which it connects to MySQL, it's also a very accessible technology—not to mention that both platforms can be obtained free of charge. If you're paying for a web hosting service, chances are good you already have access to both PHP and MySQL. For all of these reasons, PHP makes an excellent choice for an external service engine.

In Chapter 16, we provide an introductory lesson integrating Flex with PHP and MySQL using the **HTTPService** component you learned about in this chapter. However, in Chapter 16, we not only use **HTTPService** to receive data, but you also will learn how to send data using the **POST** method.

Summary

You've taken a giant leap in creating full-featured rich Internet applications and accessing data. In this chapter, you discovered how to work with data that's created within your MXML code, as well as data that's spread about the Web. You learned some basics of working with XML data, and you also learned a great deal about using **List** controls to display a series of data, including tabular data via the more advanced **DataGrid** component.

We also experimented with a third-party component to supplement the standard Flex components, which opens up a whole new world of resources you can turn to. In the process you built a simple search application using the Yahoo! ASTRA Web API, and you even created a custom **ItemRenderer** component to create a rich, interactive display for the search results.

With these skills, you can begin to approach and harness the incredible amount of data that exists on the Web. In Chapter 12, we discuss the subjects of navigation, visibility, and flow in a Flex application, and while we're at it, we'll create a Photo Gallery application to demonstrate and apply these skills as we consider them.

Creating an Application from a Database

If you're using ColdFusion, PHP, J2EE, or ASP.NET, you're in luck. Flash Builder can actually generate server-side code for you, letting you easily connect to data in a database, and even modify that data. You supply the database, and all the necessary code for creating, reading, updating, and deleting records will be written for you—even a simple UI for testing purposes.

For PHP, J2EE, or ASP.NET applications, you must have a database and a web server set up using one of these technologies. Then, all you need to do is open the Data menu and a number of options will present themselves. Once you follow the prompts, an actual Flex application will be created, containing user interface elements and service methods for accessing the host data. You can even use this as a foundation for creating an application.

CONTROLLING VISIBILITY AND NAVIGATION

“I pleased myself with the design, without determining whether I was ever able to undertake it; not but that the difficulty of launching my boat came often into my head; but I put a stop to my own inquiries into it, by this foolish answer which I gave myself, ‘Let’s first make it; I’ll warrant I’ll find some way or other to get it along, when ’tis done.’”

—Daniel Defoe,
Robinson Crusoe

IN THIS CHAPTER

- Controlling Visibility
- Navigation Components
- Creating a Photo Gallery Application
- Summary

One advantage of developing with Flex is the capacity to create dynamic web interfaces reminiscent of desktop applications. In part, this is handled by controlling the visibility of and navigation to different content areas. A company’s website might include different sections for featured products and/or services, recently completed projects, staff profiles, a blog and discussion forum, and so on. This much content cannot be presented at once on the same screen, so the site is partitioned into several different content areas.

The long-standing approach to navigating static websites requires dividing the site into several individual pages, each requiring a complete browser reload in order to view it. However, the Web 2.0 approach allows for dynamic delivery through various modules and widgets; even the main body content can be changed without refreshing the entire page.

This chapter covers the basics of navigation and flow control in Flex applications. First, we look at the **visible** and **includeInLayout** properties available on all visual elements, and then we explore the Flex navigation controls. Finally, after covering the basics, we create a photo gallery to crystallize your understanding.

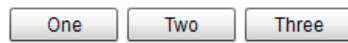


Figure 12-1. Three Button controls in an HGroup, all visible



Figure 12-2. Three Button controls in an HGroup; the second button's visible property is set to false



Figure 12-3. Three Button controls in an HGroup; the second button's includeInLayout property is set to false

Controlling Visibility

Every visual element in Flex has a **visible** property that you can toggle programmatically, and as you may have guessed, it's set to **true** by default. The **visible** property is available to every visual component, so it applies to controls and containers. That means you can set the visibility of a container, and that setting will affect each of the container's children. Let's apply this concept.

Setting **visible** to **false** makes a component invisible; however, it continues to occupy the same space. For example, if you have three buttons aligned horizontally inside an **HGroup**, setting the middle button's visibility to **false** allows it to preserve space while the first and third buttons remain separated by the same distance. On the contrary, if you wanted to truly remove the middle button from its parent container, you would set its **includeInLayout** property to **false**. Figures 12-1 through 12-3 illustrate this behavior.

Navigation Components

There might not always be enough space onscreen for everything you want to display. In such situations, you can make your content selectable using tabs. Consider how user settings are presented in most desktop applications and operating systems. Complex arrays of preferences are arranged by category—and those categories are often organized by tabs. This is just one way to handle navigation in an application.

Flex comes with a standard set of components—called *navigators*—that help you control the flow of your application by arranging visible elements into different views. Instead of presenting their children in vertical or horizontal (or other) layouts, navigators provide the means to switch between their children, showing one child at a time while hiding the others.

This section explores some of the navigator controls available to you in Flex, particularly the **TabNavigator**, **TabBar**, **ButtonBar**, **LinkBar**, and **Accordion**.

TabNavigator

The **TabNavigator** is one of the most common navigator controls, so we'll discuss it first. The **TabNavigator** takes any number of child containers and provides a tab for each. Like all navigator components, the **TabNavigator** requires its children to be container objects. Containers, as you know, group other elements into a single entity; thus, they're perfectly suited to handling thematic content.

You set a tab's text by assigning a **label** property to each container attached to the navigator control. Containers from the Halo (**mx:**) namespace have a ready-made **label** property, but Spark (**s:**) containers must be wrapped in a **NavigatorContent** component, which is used exclusively to serve the required **label** property to a navigator control. With that said, the **TabNavigator** doesn't want an array of labels for its tabs; it simply wants a group of containers that have a **label** property.

Example 12-1 establishes a **TabNavigator** with three children. Each child has a unique background color representing thematic content, and each is selected by clicking a corresponding tab. Figure 12-4 displays the running result.

Example 12-1. A **TabNavigator** with three Spark container children; each **BorderContainer** is wrapped in a **NavigatorContent** component that provides a label for a corresponding tab

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
  </s:layout>

  <mx:TabNavigator id="tabNavigator" backgroundColor="#00000">

    <s:NavigatorContent label="Red">
      <s:BorderContainer height="225" width="225"
        backgroundColor="#FF0000"
        borderColor="#000000"/>
    </s:NavigatorContent>

    <s:NavigatorContent label="Green">
      <s:BorderContainer height="225" width="225"
        backgroundColor="#00FF00"
        borderColor="#000000"/>
    </s:NavigatorContent>

    <s:NavigatorContent label="Blue">
      <s:BorderContainer height="225" width="225"
        backgroundColor="#0000FF"
        borderColor="#000000"/>
    </s:NavigatorContent>

  </mx:TabNavigator>

</s:Application>
```

NOTE

*Navigator components take container objects as their children. Containers from the Halo (**mx:**) namespace have a built-in **label** property, so they can be placed directly into navigators; however, Spark (**s:**) containers, which do not have a **label** property, must be wrapped within a **NavigatorContent** component.*



Figure 12-4. A **TabNavigator** with three children creates three tabs

In Example 12-1, the **TabNavigator** places the three Spark **BorderContainers**, each wrapped in a **NavigatorContent** component, into the same layout space; however, only the container represented by the selected tab will be visible. When a different tab is selected, its container will become visible and the others will become invisible. All navigator containers provide this approach for controlling content visibility.

As was previously mentioned, Halo (**mx:**) containers can be nested directly under a **TabNavigator**. Example 12-2 creates the same result using Halo **Canvas** containers in place of the Spark **NavigatorContent/BorderContainer** combination. Halo containers can be placed directly into the **TabNavigator** because they have the **label** property that is required to create a corresponding tab.

Example 12-2. A **TabNavigator** with three Halo container children; each Halo **Canvas** has a **label** property that's used to create a corresponding tab

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
  </s:layout>

  <mx:TabNavigator id="tabNavigator" backgroundColor="#000000">

    <mx:Canvas id="redCanvas" label="Red"
      height="225" width="225"
      backgroundColor="#FF0000"
      borderColor="#000000"/>

    <mx:Canvas id="greenCanvas" label="Green"
      height="225" width="225"
      backgroundColor="#00FF00"
      borderColor="#000000"/>

    <mx:Canvas id="blueCanvas" label="Blue"
      height="225" width="225"
      backgroundColor="#0000FF"
      borderColor="#000000"/>

  </mx:TabNavigator>

</s:Application>
```

NOTE

Remember the **Panel** container? It's represented in both namespaces. Note that its **title** property isn't the same as a **label** property; rather, its **title** is the text displayed in its own title bar. Like other containers, the Spark **Panel** will need to be wrapped in a **NavigatorContent** component in order to work with a **TabNavigator**.

TabBar and ViewStack

The **TabNavigator** inherits its features from the aptly named **ViewStack** container. In fact, the **ViewStack** is the workhorse behind the functionality of switching views, only it doesn't come with any visual cues, such as tabs, buttons, or otherwise. To get visual—or maybe we should say clickable—UI controls, you pair a **ViewStack** with a navigation control such as a **TabBar**, **ButtonBar**, or **LinkBar**, all of which can take a **ViewStack** as their **dataProvider**.

Just because the **ViewStack** is included as a **dataProvider** doesn't mean it's a nonvisual element. Rather, the **ViewStack** is a proper container that occupies layout space and responds to positioning attributes. This means the **ViewStack** can be positioned anywhere in your layout, and it responds to any navigator it's bound to.

Example 12-3 connects a **ViewStack** to a **TabBar** navigator. The navigator's **dataProvider** is bound to the **ViewStack** using the latter's **id** property. Once again, **NavigatorContent** wrappers provide **label** properties for the **TabBar**. Notice the **VGroup** container's **gap** and **horizontalAlign** properties are set to **0** and **center**, respectively, to ensure the **TabBar** appears directly above the **ViewStack** container.

Example 12-3. A **TabBar** navigator linked to a **ViewStack** container; the **TabBar**'s **dataProvider** is bound to the **ViewStack** using the latter's **id** property

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:layout>
        <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
    </s:layout>

    <s:VGroup horizontalAlign="center" gap="0">

        <s:TabBar id="tabBar" dataProvider="{viewStack}"/>

        <mx:ViewStack id="viewStack" height="225" width="225">

            <s:NavigatorContent label="Red">
                <s:BorderContainer
                    height="100%" width="100%"
                    backgroundColor="#FF0000"
                    borderColor="#000000" cornerRadius="5"/>
            </s:NavigatorContent>

            <s:NavigatorContent label="Green">
                <s:BorderContainer
                    height="100%" width="100%"
                    backgroundColor="#00FF00"
                    borderColor="#000000" cornerRadius="5"/>
            </s:NavigatorContent>

            <s:NavigatorContent label="Blue">
                <s:BorderContainer
                    height="100%" width="100%"
                    backgroundColor="#0000FF"
                    borderColor="#000000" cornerRadius="5"/>
            </s:NavigatorContent>

        </mx:ViewStack>

    </s:VGroup>

</s:Application>
```

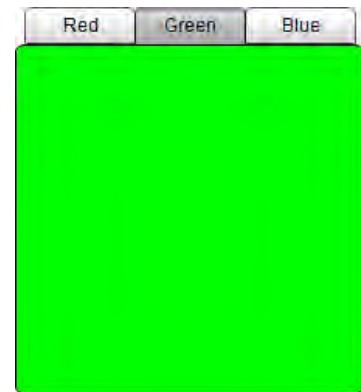


Figure 12-5. A **TabBar** navigator and **ViewStack** nested in a **VGroup**

Figure 12-5 shows how it looks when launched.

Why would you want to pair a **ViewStack** with a **TabBar** when you can use a **TabNavigator**? Well, a **ViewStack** gives you the option of separating the navigator from the content, which creates more flexibility for styling and skinning; this would allow you to frame the content with a special border, for example.

So, what if you want navigation functionality without the “tab” look? In this case, you might use a **ButtonBar** instead of a **TabBar**.

ButtonBar and ViewStack

Example 12-4 demonstrates use of the **ButtonBar** with a **ViewStack**. Unlike previous examples that set both components within a **VGroup**, this code uses a regular **Group** container, **BasicLayout**, and constraints to seat the **ButtonBar** directly below the **ViewStack**. To create this look, we assigned the **ButtonBar** navigator’s **bottom** property a value of **0** and the **ViewStack** container’s **bottom** property a value of **21**, which equals the height of the **ButtonBar**. We learned the height of the **ButtonBar** by running a **trace()** on its **height** property. In particular, we’re emphasizing the option of arranging a navigator and a **ViewStack** according to specifications for a more unique layout. Figure 12-6 shows the result.

Example 12-4. *Using layout properties to arrange a ButtonBar and a ViewStack so that the ButtonBar appears below the ViewStack*

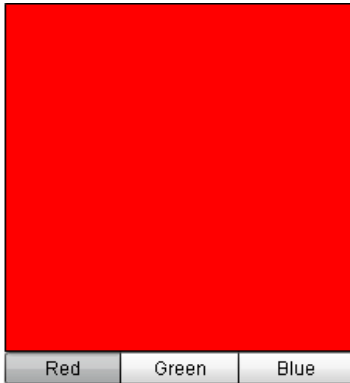


Figure 12-6. A **ButtonBar** and **ViewStack** arranged in a **BasicLayout**

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:layout>
        <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
    </s:layout>

    <s:Group>

        <s:layout><s:BasicLayout/></s:layout>

        <mx:ViewStack id="viewStack" bottom="21" height="225" width="225">

            <s:NavigatorContent label="Red">
                <s:BorderContainer height="100%" width="100%"
                    backgroundColor="#FF0000" borderColor="#000000"/>
            </s:NavigatorContent>

            <s:NavigatorContent label="Green">
                <s:BorderContainer height="100%" width="100%"
                    backgroundColor="#00FF00" borderColor="#000000"/>
            </s:NavigatorContent>

            <s:NavigatorContent label="Blue">
                <s:BorderContainer height="100%" width="100%"
                    backgroundColor="#0000FF" borderColor="#000000"/>
            </s:NavigatorContent>

        </mx:ViewStack>

        <s:ButtonBar id="buttonBar" dataProvider="{viewStack}"
            bottom="0" width="100%"/>

    </s:Group>

</s:Application>
```

LinkBar and ViewStack

You create a **LinkBar** using the same syntax you used to create the **TabBar** and the **ButtonBar**. However, **LinkBar** navigators have the look and feel of hyperlinks, so they're classic and familiar.

When we discussed the **TabBar**, we mentioned the **ViewStack** container responds to any navigator it's bound to. This means you can bind the **ViewStack** to multiple navigators. Example 12-5 attaches both a **TabBar** and a **LinkBar** to the same **ViewStack** with the intention of emulating a classic look. See the result in Figure 12-7.

Example 12-5. *Connecting a LinkBar navigator to a ViewStack that is already bound to a TabBar navigator*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:layout>
        <s:VerticalLayout verticalAlign="middle" horizontalAlign="center"/>
    </s:layout>

    <s:VGroup horizontalAlign="center" gap="0">

        <s:TabBar id="tabBar" dataProvider="{viewStack}"/>

        <mx:ViewStack id="viewStack" height="225" width="225">

            <s:NavigatorContent label="Red">
                <s:BorderContainer height="100%" width="100%"
                    backgroundColor="#FF0000" borderColor="#000000"/>
            </s:NavigatorContent>

            <s:NavigatorContent label="Green">
                <s:BorderContainer height="100%" width="100%"
                    backgroundColor="#00FF00" borderColor="#000000"/>
            </s:NavigatorContent>

            <s:NavigatorContent label="Blue">
                <s:BorderContainer height="100%" width="100%"
                    backgroundColor="#0000FF" borderColor="#000000"/>
            </s:NavigatorContent>
        </mx:ViewStack>

        <mx:LinkBar id="linkBar" dataProvider="{viewStack}"/>

    </s:VGroup>
</s:Application>
```

NOTE

Similarly, you can connect a **TabNavigator** to a **LinkBar** by binding the **dataProvider** of the **LinkBar** to the **id** of the **TabNavigator**.

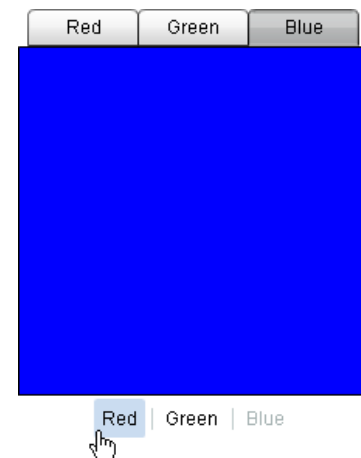


Figure 12-7. A **TabBar** and a **LinkBar** bound to the same **ViewStack**

Controlling the ViewStack with ActionScript

NOTE

For all navigator containers, the `selectedIndex` default value is 0, and the `selectedChild` default is the first child.

Besides the capacity to bind with navigator controls, the **ViewStack** container can also be controlled programmatically through its `selectedIndex` and `selectedChild` properties. The `selectedIndex` property takes an integer value and is valid as long as the parent **ViewStack** has a container in the referenced index position. Consider the **ViewStack** from the previous example. You could use either inline ActionScript or a named function within a Script/CDATA block to control the visible content; we consider both approaches in this section.

Controlling navigation using inline ActionScript

The **TabBar**, **ButtonBar**, and **LinkBar** do not *require* a **ViewStack** as their `dataProvider`; they can also populate their menus using an **ArrayCollection**. Therefore, if you wanted to control navigation using MXML and inline ActionScript, Example 12-6, which uses a **ButtonBar** to demonstrate this scenario, would suffice. The task of navigating is handled inline using the navigator's `click` event. Figure 12-8 shows the result.

Example 12-6. *Populating a ButtonBar using an ArrayCollection and controlling navigation using inline script*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Declarations>
        <s:ArrayCollection id="colorsMenuAC" source="[Red, Green, Blue]" />
    </fx:Declarations>

    <s:layout><s:BasicLayout/></s:layout>

    <s:ButtonBar id="buttonBar" dataProvider="{colorsMenuAC}"
        click="{viewStack.selectedIndex = buttonBar.selectedIndex}"
        top="10" left="10"/>

    <mx:ViewStack id="viewStack" height="100%" width="100%"
        top="31" bottom="10" left="10" right="10">

        <s:NavigatorContent label="Red">
            <s:BorderContainer height="100%" width="100%"
                backgroundColor="#FF0000" borderColor="#000000" />
        </s:NavigatorContent>

        <s:NavigatorContent label="Green">
            <s:BorderContainer height="100%" width="100%"
                backgroundColor="#00FF00" borderColor="#000000" />
        </s:NavigatorContent>

    </mx:ViewStack>
</s:Application>
```

NOTE

As you can see in Figure 12-8, Example 12-6 uses constraints on the **ButtonBar** and the **ViewStack** to create a more practical layout.


```

<s:NavigatorContent label="Blue">
    <s:BorderContainer height="100%" width="100%"
        backgroundColor="#0000FF" borderColor="#000000"/>
</s:NavigatorContent>

</mx:ViewStack>

</s:Application>

```

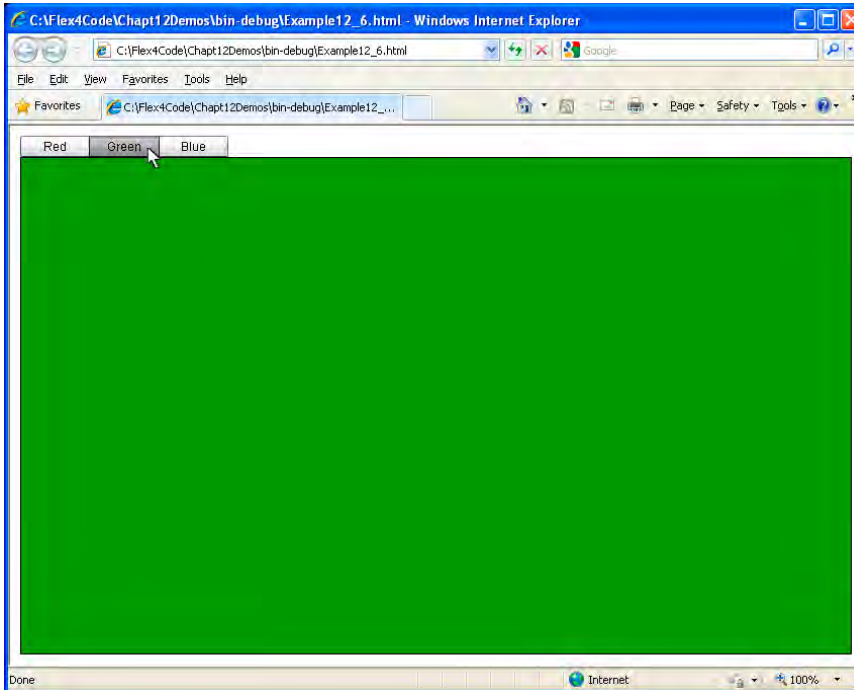


Figure 12-8. Controlling a ViewStack with inline ActionScript

This is a neat trick to remember. You'll need to make sure the sequence of menu items in the **ArrayCollection** matches the order of nested containers in the **ViewStack**; otherwise, you'll have mismatched links. Using the same concept, you could even bind the **selectedIndex** property of a **List** or a **DataGrid** to the **selectedIndex** of a **ViewStack**, in which case the **List** would handle the navigation.

Controlling navigation using named functions

Unlike the **selectedIndex**, which refers to **ViewStack** containers by their nested order, the **ViewStack** container's **selectedChild** property refers to containers by their unique **id** property. By setting the **selectedChild**, you don't have to worry about the nested order of containers; instead, you can decisively select a container by calling its **id**.

NOTE

You can load containers of a **ViewStack** by explicitly calling their index positions using a statement like the following, which calls the Red theme of the **ViewStack** used in Example 12-6:

```
viewStack.selectedIndex = 0;
```

Index positions begin at 0 and increment by 1 for every instance beyond the first. So, if you wanted to directly reference the Green theme, you would use:

```
viewStack.selectedIndex = 1;
```

And if you wanted the Blue theme:

```
viewStack.selectedIndex = 2;
```

WARNING

A **ViewStack** container's **selectedChild** property references a nested container's **id** property, not its **label**, which might have been your first guess.

In Example 12-7, we contrast both selection approaches, using both the `selectedIndex` and `selectedChild` properties to control navigation, so look for the differences. As for the named function controlling the navigation, we used a `Switch..Case` block that takes the `ButtonBar` navigator's `selectedItem`, which returns its `label`—either Red, Green, or Blue—and uses that value to decide which container is visible. Finally, we made the `ViewStack` children a little more complex to emphasize that these containers have unique content. See the result in Figure 12-9.

Example 12-7. Controlling navigation using a `Switch..Case` block inside a named function

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <fx:Script>
    <![CDATA[
      private function onButtonBarClick():void{
        switch (buttonBar.selectedItem){
          case "Red":
            // Access the Red theme using its index position
            viewStack.selectedIndex = 0;
            break;
          case "Green":
            // Access the Green theme using its id
            viewStack.selectedChild = greenTheme;
            break;
          case "Blue":
            // Access the Blue theme using its id
            viewStack.selectedChild = blueTheme;
            break;
          default:
            break;
        }
      }
    ]]>
  </fx:Script>

  <fx:Declarations>
    <s:ArrayCollection id="colorsMenuAC" source="[Red, Green, Blue]" />
  </fx:Declarations>

  <s:layout><s:BasicLayout/></s:layout>

  <s:ButtonBar id="buttonBar" dataProvider="{colorsMenuAC}"
    click="onButtonBarClick()" top="10" left="10"/>
```

```

<mx:ViewStack id="viewStack" height="100%" width="100%"
    top="31" bottom="10" left="10" right="10">

    <s:NavigatorContent id="redTheme" label="Red">
        <s:Group height="100%" width="100%">
            <s:Rect height="100%" width="100%">
                <s:fill>
                    <s:SolidColor color="#FF0000"/>
                </s:fill>
            </s:Rect>
            <s:Label top="10" left="10" fontWeight="bold"
                text="This is the Red theme."/>
            <s:Button label="Button One" bottom="10" right="10"/>
        </s:Group>
    </s:NavigatorContent>

    <s:NavigatorContent id="greenTheme" label="Green">
        <s:Group height="100%" width="100%">
            <s:Rect height="100%" width="100%">
                <s:fill>
                    <s:SolidColor color="#009921"/>
                </s:fill>
            </s:Rect>
            <s:Label top="10" left="10" color="#FFFFFF"
                text="This is the Green theme." fontWeight="bold"/>
            <s:Button label="Button Two" bottom="10" right="10"/>
        </s:Group>
    </s:NavigatorContent>

    <s:NavigatorContent id="blueTheme" label="Blue">
        <s:Group height="100%" width="100%">
            <s:Rect height="100%" width="100%">
                <s:fill>
                    <s:SolidColor color="#0000FF"/>
                </s:fill>
            </s:Rect>
            <s:Label top="10" left="10" color="#FFFFFF"
                text="This is the Blue theme." fontWeight="bold"/>
            <s:Button label="Button Three" bottom="10" right="10"/>
        </s:Group>
    </s:NavigatorContent>

</mx:ViewStack>

</s:Application>

```

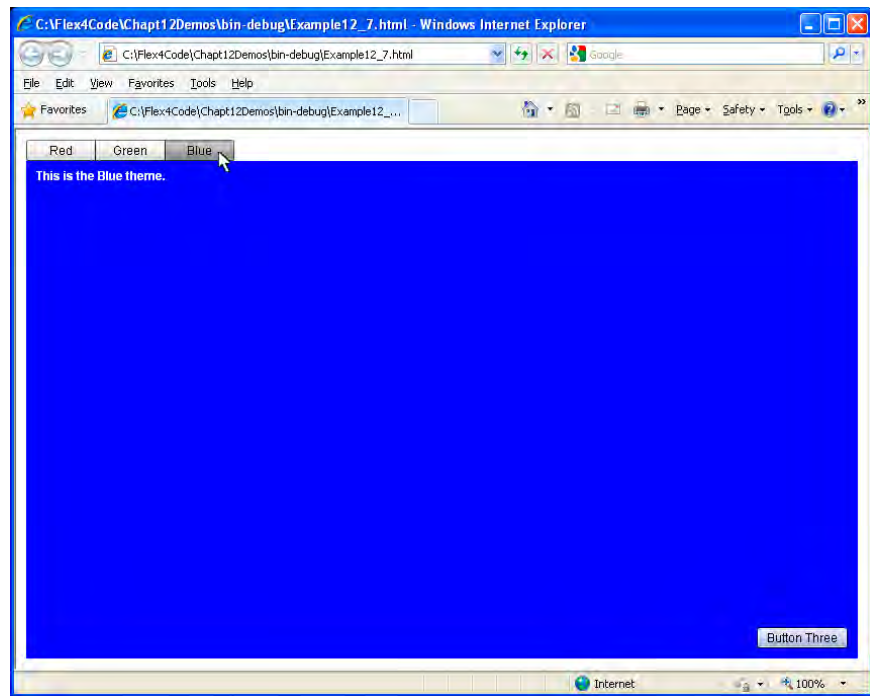


Figure 12-9. Controlling a ViewStack with a named function

Controlling navigation using ActionScript can be useful for forms that are divided into categories. You could use this approach to lead people through the form, providing them with a multistep approach to completing the form as well as giving them a sense of progress. After all, who wants to see a huge, scrolling form? It's uninviting to say the least. By partitioning a form into categories, you can make it appear small at first glance. If you were interested in creating such an approach, you could provide a Next button that, when clicked, incremented the `selectedIndex` of your `ViewStack` by 1.

The next navigator we introduce, the **Accordion**, is perfectly suited for this tactic.

The Accordion Navigator

The **Accordion** works by stacking its contents' title bars one above the other. When a user clicks on one of the title bars, the **Accordion** literally slides into the selected theme, shutting the prior theme in the same motion. In code, the **Accordion** closely resembles the **TabNavigator**. Since both controls' declarations are so similar, we'll take an opportunity to step out of context and briefly introduce you to a simple Flex effect—**Bounce**—that makes the **Accordion** navigator fun to use.

In Example 12-8, a Script/CDATA block is included solely to import the desired effect class, `mx.effects.easing.Bounce`, and within the MXML, the effect is managed by two simple attribute properties—`openEasingFunction` and `openDuration`. The first property specifies the exact method to call from the effect class, in this case `Bounce.easeOut`, and the second property establishes how much time the effect should consume to complete its routine. Figure 12-10 shows the result.

Example 12-8. An Accordion with nested Spark content; the easing effect makes the Accordion appear to “bounce” as it settles into the next selection

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import mx.effects.easing.Bounce;
        ]]>
    </fx:Script>

    <mx:Accordion id="accordion" horizontalCenter="0" verticalCenter="0"
        openEasingFunction="{Bounce.easeOut}" openDuration="1000">

        <s:NavigatorContent id="redTheme" label="Red">
            <s:BorderContainer height="225" width="225"
                backgroundColor="#FF0000"
                borderColor="#000000"/>
        </s:NavigatorContent>

        <s:NavigatorContent id="greenTheme" label="Green">
            <s:BorderContainer height="225" width="225"
                backgroundColor="#00FF00"
                borderColor="#000000"/>
        </s:NavigatorContent>

        <s:NavigatorContent id="blueTheme" label="Blue">
            <s:BorderContainer height="225" width="225"
                backgroundColor="#0000FF"
                borderColor="#000000"/>
        </s:NavigatorContent>

    </mx:Accordion>
</s:Application>
```

WARNING

The `Accordion..Bounce` example is based on an example provided through Adobe’s online help system (we used [bit.ly](http://bit.ly/cwvSvK) to shorten the link): <http://bit.ly/cwvSvK>.

Curiously, code completion did not want to offer assistance in finding the `Bounce` effect methods, `easeOut` and `easeIn`. This might be a bug within the Flash Builder Beta release we used to write this book, but if you run into problems yourself, take heart in knowing that you’re not making a mistake. Just enter the code as you see it here and run it. It will work.

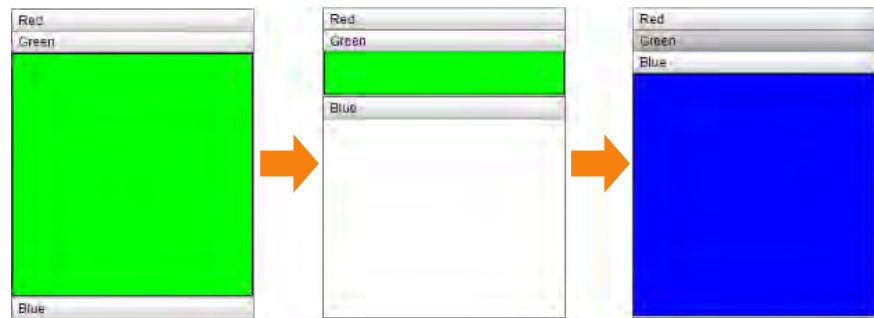


Figure 12-10. A bouncy Accordion slamming the door on the green theme

The **Accordion** navigator is desirable for controlling sequential flow through a set of related themes. On the other hand, it can become cumbersome if there are too many steps or items. So have fun with it, but use it carefully.

Creating a Photo Gallery Application

Now we put the navigation components to good use by building a photo gallery application. Chances are you've encountered an online photo gallery somewhere—news websites, Flickr, social media sites—so you likely have some idea of how a photo gallery should flow. Modern implementations often provide multiple options for viewing photos. One obvious example is the option to see filenames or image thumbnails. With Flex, we can easily create something similar to show off photos in our web applications, and that's exactly what we'll do in the last section of this chapter.

So, to get started, open Flash Builder and create a new project called *PhotoGallery*.

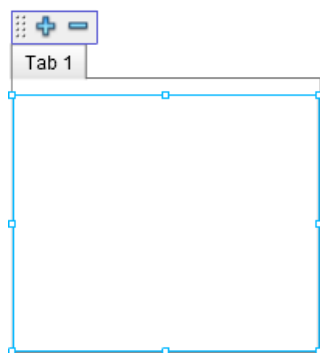


Figure 12-11. Adding a TabNavigator in Design mode; the tools above the control help you move it around the layout as well as add (+) or remove (–) additional containers

Adding Multiple Tab Views

For the first step, jump into Design mode and drag a **TabNavigator** control into your new application. This automatically creates a **TabNavigatorContent** wrapper as its child. Before moving ahead, give the **TabNavigator** the following properties: **width="200"**, **left**, **top**, and **bottom="10"**.

When you select the **TabNavigator**, a toolbar will display directly above the control with a “gripper,” or anchor handle, as well as a plus and minus sign (Figure 12-11). The gripper provides the means of moving the **TabNavigator** around in your layout, and the plus (+) and minus (–) signs let you add and remove containers. If you want to remove a container while in Design mode, click on the container's tab to select it, and then click the minus symbol.

Clicking the plus symbol will open a dialog box prompting you to select a container type to add to the **TabNavigator**. Options include **NavigatorContent**, **Module**, **Form**, and so on. The dialog also allows you to add the container's tab label. So, using either the dialog box or by switching into Source mode, add two **NavigatorContent** children in the **TabNavigator** (Figure 12-12). Give them the labels "Photo List" and "Thumbnails". While you're at it, remove "Tab 1", which was created by default when you added the **TabNavigator** in Design mode.

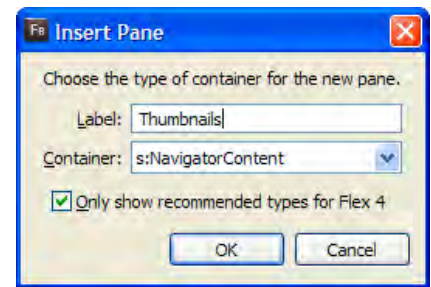


Figure 12-12. Adding a **NavigatorContent** container in Design mode

Selecting Navigator Containers in Design Mode

When manipulating navigation components in Design mode, it can be difficult to know whether you selected the navigation component or its children containers. This problem can be solved by using the Outline pane, which is helpful for selecting components in tight situations; see Figure 12-13.

If you want to work exclusively within the layout, though, you can select the **TabNavigator** by clicking its gripper next to the plus and minus signs. The gripper is useful for selecting the **TabNavigator** as well as dragging it around the layout. If you want to select an individual container, just select its tab.

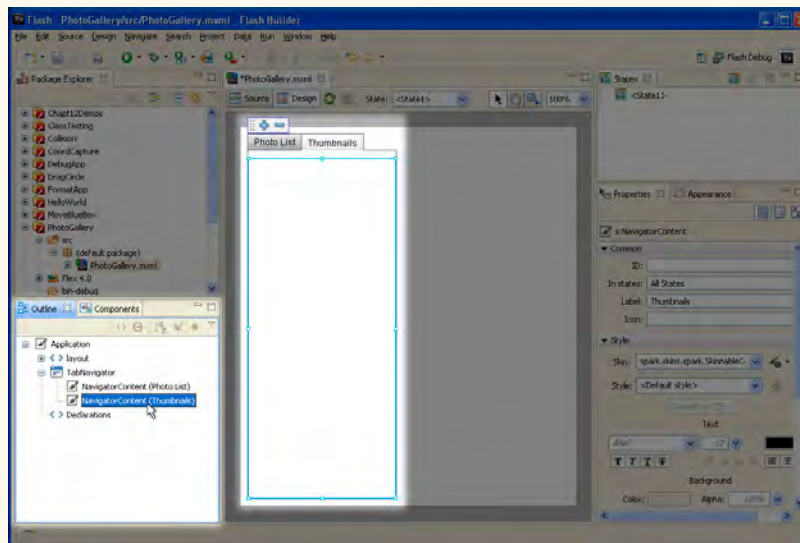


Figure 12-13. Selecting the **Thumbnails** view in the Outline pane

With both **NavigatorContent** wrappers in place, add a **List** control to each and set both their **height** and **width** properties to 100% so they will consume all available space within the **TabNavigator**. Give the lists **id** values of **photoList** and **thumbList**. At this point, your code should resemble Example 12-9.

Example 12-9. *The beginnings of the PhotoGallery application*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <s:layout>
    <s:BasicLayout/>
  </s:layout>

  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>

  <mx:TabNavigator left="10" top="10" bottom="10" width="200">

    <s:NavigatorContent label="Photo List" width="100%" height="100%">
      <s:List id="photoList" height="100%" width="100%" />
    </s:NavigatorContent>

    <s:NavigatorContent label="Thumbnails" width="100%" height="100%">
      <s:List id="thumbList" height="100%" width="100%" />
    </s:NavigatorContent>
  </mx:TabNavigator>

</s:Application>
```

Populating the Photo Gallery with XML

We're ready to add some photo data to the **List** components. We'll handle the task using XML. While we encourage you to customize the XML to point to your own images, we provided both an XML file and the linked images through the companion website, which you are welcome to use. For tips on using your own images, see the sidebar titled "XML Options" on page 259.

Creating the photos.xml file

Carrying on with the example, create a new file called *photos.xml* in the **PhotoGallery** application's *src* folder. To create an XML file, right-click the desired directory (*src*) and select New→File. When prompted, name the file *photos.xml*. Make sure to include the *.xml* extension. Once the file is created and open for editing, add the XML in Example 12-10.

Example 12-10. *C:\Flex4Code\PhotoGallery\src\photos.xml*

```

<photos>
  <photo credit="Sharif Zawaideh" title="Yawning Camel"
    thumb="http://www.learningflex4.com/photos/camel_th.jpg"
    image="http://www.learningflex4.com/photos/camel.jpg"/>
  <photo credit="Sharif Zawaideh" title="Crowdy Head Lighthouse"
    thumb="http://www.learningflex4.com/photos/lighthouse_th.jpg"
    image="http://www.learningflex4.com/photos/lighthouse.jpg"/>
  <photo credit="Sharif Zawaideh" title="Uluru"
    thumb="http://www.learningflex4.com/photos/uluru_th.jpg"
    image="http://www.learningflex4.com/photos/uluru.jpg"/>
  <photo credit="Sharif Zawaideh" title="Devil's Marbles"
    thumb="http://www.learningflex4.com/photos/marble_th.jpg"
    image="http://www.learningflex4.com/photos/marble.jpg"/>
  <photo credit="Grant Tomlins" title="Cliffside"
    thumb="http://www.learningflex4.com/photos/cliff_th.jpg"
    image="http://www.learningflex4.com/photos/cliff.jpg"/>
  <photo credit="Grant Tomlins" title="Sailing"
    thumb="http://www.learningflex4.com/photos/sailing_th.jpg"
    image="http://www.learningflex4.com/photos/sailing.jpg"/>
  <photo credit="Grant Tomlins" title="Kinsale"
    thumb="http://www.learningflex4.com/photos/kinsale_th.jpg"
    image="http://www.learningflex4.com/photos/kinsale.jpg"/>
  <photo credit="David Baclian" title="Waiting"
    thumb="http://www.learningflex4.com/photos/waiting_th.jpg"
    image="http://www.learningflex4.com/photos/waiting.jpg"/>
  <photo credit="John Van Every" title="Washington Monument"
    thumb="http://www.learningflex4.com/photos/washington_th.jpg"
    image="http://www.learningflex4.com/photos/washington.jpg"/>
</photos>

```

The XML code is a simple list of photo nodes, each having a **credit**, **title**, **thumb**, and **image** attribute. Of course, the **title** attribute is the title of the image, and the **thumb** and **image** attributes contain URLs pointing to the corresponding resource.

XML Options

If you don't want to type out all the XML, you can use the XML file we made available on the companion website. To use this option, jump down to the section "Connect to the XML with an HTTPService component" on page 260, and when you create the **HTTPService** component, point its **url** property to this address: <http://www.learningflex4.com/photos/photos.xml>.

Furthermore, to use your own images, you should copy them to a new directory under the **PhotoGallery** application's **src** folder and change the URLs in the XML file to point to your images using links relative to the application. For instance, if you have both an image named *whitedog.jpg* and its thumbnail (*whitedog_th.jpg*) in a folder named *myphotos*, that node would be:

```

<photo credit="Your Name" title="White Dog"
  thumb="myphotos/whitedog_th.jpg"
  image="myphotos/whitedog.jpg"/>

```

As you learned previously, any files you place in your **src** folder will be copied to the **bin-debug** directory. So if you add an entire folder of images, it will be copied to the **bin-debug** directory, and your application will load images from there when run.

Connect to the XML with an HTTPService component

To connect to the XML file, create an **HTTPService** component to call in the data. Because **HTTPService** is a nonvisual component, you need to be in Source mode to add it. Make sure to place it in the Declarations section of your application code. Configure its properties as shown in Example 12-11, and remember, the **id** value is essential to using this component.

Example 12-11. Adding an **HTTPService** component to the Declarations section

```
<fx:Declarations>

    <s:HTTPService id="photoService" url="photos.xml" resultFormat="e4x"/>

</fx:Declarations>
```

You'll want to prompt the service as soon as the application loads. You can trigger this using the **Application** container's **applicationComplete** event listener and inline ActionScript to call the **HTTPService** component's **send()** method, as demonstrated in Example 12-12.

Example 12-12. Calling the **HTTPService** component's **send()** method with the **Application** container's **applicationComplete** event

```
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    applicationComplete="photoService.send()">
```

For the next step, we need to properly absorb the XML data before we can connect it to our **List** components. If you remember from the previous chapter, the **dataProvider** of a **List** control takes a list of items. We'll handle that requirement by creating another Declarations component, **XMLListCollection**, to receive the incoming data and serve it as a list. With that said, create an **XMLListCollection** below the **HTTPService** component in the Declarations section and bind its **source** property to **photoService.lastResult.photo**, as in Example 12-13.

Example 12-13. Creating an **XMLListCollection** and binding it to the **HTTPService**'s **lastResult**

```
<s:XMLListCollection id="photoXMLList"
    source="{photoService.lastResult.photo}"/>
```

The statement **photoService.lastResult.photo** points to the list of **<photo/>** nodes in the returning XML file. We're finally ready to display something in our lists.

For **photoList**, set its **dataProvider** to **{photoXMLList}** and its **labelField** to **@title**, as shown in Example 12-14. Because the title of the image is included as an attribute in XML, it must be accessed by the E4X expression **@title**.

Example 12-14. *Linking photoList to the XMLListCollection*

```
<s:NavigatorContent label="Photo List">
  <s:List id="photoList" height="100%" width="100%"
    dataProvider="{photoXMLList}" labelField="@title"/>
</s:NavigatorContent>
```

Configuring the **labelField** as we did here informs the **List** of what to display. Don't worry about the **thumbList** control just yet; we'll address it a little later.

Displaying External Images

The next step is displaying the images, and because the **Image** control was designed just for this purpose, it's really easy to set up. Simply adding an **Image** control to your application and setting its **source** property to a URL will cause it to load the image. Any time the **Image** control's **source** property changes, such as during a data binding, the image will reload.

So place an **Image** control in your application. If you're working in Design mode, drag it anywhere to the right of the **TabNavigator**, but if you're working in Source mode, add it below the code for the **TabNavigator**. Give the **Image** control an **id** of **photoImage**, and set its **source** property to **{photoList.selectedItem.@image}**. This will bind the **source** to the corresponding URL attribute of the XML node that is currently selected in the **List** control.

If no size is set on the **Image** control, it will resize itself to the actual size of the source image. So if the image or photo loaded is 300 pixels wide and 100 pixels tall, the **Image** will set its own width to 300 and its height to 100. Although this might be the desired behavior in some applications, it's often best to know in advance the size of the **Image** control so you can plan your layout better. Setting an explicit size on the **Image** control will cause it to scale the loaded photo to fit within its bounds. So you can set the **Image** control to an explicit **height** and **width**, but it would be even better to use constraints to anchor the image to the edges of the application. With a constraints-based layout, users can resize their browsers and the **Image** control will resize to fit the new dimensions. Knowing this, configure your **Image** to use the following layout constraints: **left="220"**, **top**, **bottom**, and **right="10"**.

You also want to set the **horizontalAlign** and **verticalAlign** properties of the **Image** to **center** and **middle**, respectively, so the image will automatically center within the **Image** control's bounds. Altogether, you should now have the **Image** component tag shown in Example 12-15.

Example 12-15. *The Image component declaration*

```
<mx:Image id="photoImage" source="{photoList.selectedItem.@image}"
  horizontalAlign="center" verticalAlign="middle"
  left="220" top="10" bottom="10" right="10"/>
```

At this point, selecting items in **photoList** will cause the **Image** control to load an image. Go ahead and run the application; it should resemble Figure 12-14.

WARNING

*When running the **PhotoGallery** at this stage, you need to select one of the **List** items before the image will load.*

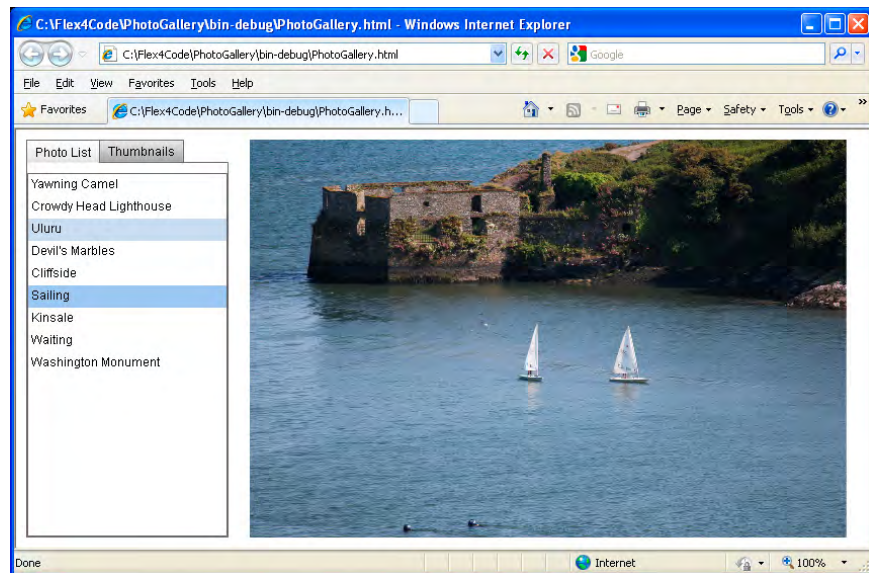


Figure 12-14. The photo gallery in a List view

Monitoring the Loading Progress of Images

You may have noticed that it takes a moment to load some of the images. If a connection is slow, some users might become confused if they expect to click a **List** item and see an image load immediately. If the image takes a while to load, the person may assume something went wrong and click another item. Of course, this only makes matters worse, as the loading process must begin anew.

To provide better feedback in your application, you can use a **ProgressBar** to monitor the progress of loading images. You can use a **ProgressBar** for many purposes, but it works perfectly in combination with an **Image** control. All you have to do is bind the **ProgressBar** component's **source** property to the **id** of the **Image** control, and it just works.

So, to make the **PhotoGallery** more user-friendly, add a **ProgressBar** control with an **id** of **progressBar** to your application. You need to be in Source mode, and you should place the **ProgressBar** directly below the **Image** control. Set the component's **source** property to **{photoImage}**. Additionally, assign the following properties: **width="100"**, **bottom="15"**, and **right="30"**. Now when an item is selected in the **List**, that image will load, and the progress bar will display the loading progress.

With these properties, the progress bar will remain in view after the image is fully loaded. That's not a desired behavior; once the image loads, we want the progress bar to disappear. It would be better if the progress bar appeared and disappeared as needed. Fortunately, that's easy to accomplish using the **visible** property and a couple of events built into the **Image** control.

The **Image** control has two events pertinent to this scenario: **open** and **complete**. The **open** event fires when a source begins loading, and the **complete** event fires when loading is complete. We'll use these events to manage **ProgressBar** visibility.

First, set the **ProgressBar** to be initially invisible by setting its **visible** property to **false**. Next, enable the **open** and **complete** listeners on the **Image** control to toggle the **ProgressBar** component's visibility. On **open**, set **progressBar.visible** to **true**, and on **complete**, set **progressBar.visible** to **false**. By now, the code for these two controls should resemble Example 12-16.

Example 12-16. *The code for the PhotoGallery's Image and ProgressBar components*

```
<mx:Image id="photoImage" source="{photoList.selectedItem.@image}"
  horizontalAlign="center" verticalAlign="middle"
  left="220" top="10" bottom="10" right="10"
  open="progressBar.visible = true"
  complete="progressBar.visible = false"/>

<mx:ProgressBar id="progressBar" source="{photoImage}"
  width="100" bottom="15" right="30"
  visible="false"/>
```

Now you have a more responsive application because it informs people that their clicks are working. A little feedback goes a long way!

Browser Cache and Loading Progress

Assuming you loaded some images when you first ran the **PhotoGallery**, or if you are using local images that you supplied yourself, you won't see the **ProgressBar** change from anything other than 100%. This is because that image is either local or it has been cached, meaning a copy of the image has been stored by your browser for faster retrieval. It's therefore instantly loaded, and the **ProgressBar** doesn't display any progress, because there isn't any—either it's loaded or it's not.

The browser is responsible for caching, and it caches resources as they are brought in from the Web. In fact, the browser can cache SWF files. You'll notice this if you use Flex applications on the Web. The first time you load the application, a progress bar will show, and it will take a moment for the application to load; the next time you visit the application, it will load faster.

Caching is usually the desired behavior because it facilitates faster site loading. Most browsers provide a means of disabling or emptying the cache, which is helpful to developers. If you were to empty your cache and try loading the **PhotoGallery** images again, you would see the progress bar monitoring the loading process.

Note that clearing your browser's cache can have adverse effects. You might see a decrease in speed when browsing sites you frequent because those assets are no longer cached. When developing, it might help to use one browser for daily activities and another browser for development, letting you clear the cache as necessary.

NOTE

If you're looking for a great way to connect to all kinds of photos, you can use the *ActionScript 3.0 API* available from Adobe that lets you interact with Flickr, the popular photo-sharing site. This API includes features for searching and displaying photos as well as uploading and tagging them. This API requires knowledge of *ActionScript*, but if you're interested in taking the **PhotoGallery** to the next level, get the code at <http://code.google.com/p/as3flickrlib>.

NOTE

The **ItemRenderer** you're creating is actually a class; therefore, it should be named using *UpperCamelCase*. It's good practice to name classes using this convention. Meanwhile, class instances, which are object variables declared in your code, should be named using *lowerCamelCase*.

NOTE

If you look back at the **MXML** for **thumbList**, it should have gained a reference to the **ItemRenderer** you just created:

```
<s:List id="thumbList"
  height="100%" width="100%"
  dataProvider="{photoXMLList}"
  itemRenderer="ThumbItemRenderer"/>
```

Creating the Thumbnails

In this section, we finish the incomplete **Thumbnails** view. Setting the **labelField** was enough to render the image title in the **photoList** control, but for the **thumbList** control, we want to display a thumbnail representation, and that's slightly more complex. To do that, we need to create an **ItemRenderer** that uses another **Image** control. In Chapter 11, we learned that we can use *Flash Builder* to help us start an **ItemRenderer** component, and we'll repeat that here.

First, let's add a **dataProvider** to the **thumbList** control. Within the **thumbList** **MXML**, right after the **id** property, add **dataProvider="{photoXMLList}"**. Follow the **dataProvider** attribute with a carriage return and a space, and start typing **itemRend**. You should receive code assistance; when you do, hit **Tab** followed by **Enter/Return** to select **itemRenderer**. You should then be presented with a second code hint; choose "Create Item Renderer..."

This sequence should bring a *New MXML Item Renderer* dialog into view. When the dialog appears, provide the name *ThumbItemRenderer*. Leave the **Package** field blank to save the **ItemRenderer** class in the project *src* directory. *Flash Builder* will automatically add the *.mxm* file extension when you select **Finish**.

When you emerge into the code editor, create the **ItemRenderer** provided in Example 12-17. Remember, an **ItemRenderer** is merely a composite component that a **List** control will use to load multiple data types as a single item in the **List**. The component layout of the following **ItemRenderer** should not present any surprises, but we did emphasize the data bindings responsible for identifying values in the source XML. Notice also that we're using the character sequence **** to create a line break in the **toolTip** property, making it possible to cite both the image title and the photographer credit in the **toolTip**.

Example 12-17. C:\Flex4Code\PhotoGallery\src\ThumbItemRenderer.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<s:ItemRenderer
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  autoDrawBackground="true">

  <s:VGroup width="180" horizontalAlign="center" verticalAlign="middle">

    <s:Label text="{data.@title}"/>

    <mx:Image height="70" width="100"
      horizontalAlign="center"
      source="{data.@thumb}"
      toolTip="Title: {data.@title} &#13;Credit: {data.@credit}"/>

  </s:VGroup>

</s:ItemRenderer>
```


If you run the application now, the Thumbnails tab will display an attractive assortment of thumbnails; however, clicking the thumbnails still won't do anything.

Syncing Two Lists

You can bind `thumbList` to the main `Image` control's `source` property using several different techniques. One option is to use `<fx:Binding/>` tags to create a multisource binding to the image. Doing this, you could bind the `Image` to both `List` controls; then, when either `photoList` or `thumbList` fire a selection change, the `Image` control's `source` will update. However, this isn't the best scenario, because when someone switches between the two views, each `List` might have a different selection. For example, if you clicked the third item in `thumbList` and that image loaded, and then you switched to `photoList`, that `List` might have a different item selected. In other words, the two lists would be out of sync.

Instead of binding the `Image` control to both lists, just bind it to `photoList`. To sync the two lists, add an inline `change` attribute to each of them. For the `thumbList`, have it update `photoList` whenever `thumbList` changes, and have `photoList` update `thumbList` whenever `photoList` changes. Then, when the `thumbList` item selection changes, it will call `photoList` to imitate that selection, and `photoList`, in turn, will continue causing the `Image` control to update its `source`.

Our proposed method does have one limitation regarding the manner in which navigator containers create their children. To make the application initialize faster, all views are *not* created at first. That is, only the first tab's content is initialized when the application is loaded. The other tab's content isn't created until that view is called by selecting its tab. This isn't a problem for applications that have only two tabs, but for applications with many tabs, the initial startup would be degraded without this feature. Because of this behavior, our `List` controls won't sync properly until the application user initializes the second view. The `photoList` control's `change` event would fire and attempt to update `thumbList`, but since the `thumbList` isn't yet created, nothing would happen.

To get around this problem, navigator containers have a property called `creationPolicy` that allows you to override this default behavior. The property can accept one of four values: `all`, `auto`, `queued`, or `none`. The default is `auto`, which operates as we just explained, creating only the initial view based on the 0 index. Setting `creationPolicy` to `all` creates all views at once, whereas setting it to `queued` creates all child containers and then the child containers' children in a sequence. The policy `none` prevents creating any views; it's an advanced policy that requires the developer to initialize all views using code.

WARNING

You might be asking, "Why not bind the `thumbList` control's `selectedIndex` to the `photoList` control's `selectedIndex`, and vice versa, creating a two-way binding?" The answer is that a two-way binding in this situation might create a recursion problem, causing your application's performance to degrade substantially.

NOTE

The default `creationPolicy` setting of `auto` is generally your best bet for performance reasons because it creates its views as necessary.

After considering our options, let's set the **creationPolicy** property of the **TabNavigator** to **all**. This will ensure both views are ready at startup and the selections will sync as expected. Example 12-18 has the final code for the **TabNavigator** and the two **List** controls; we emphasized the changes you should make in this section.

Example 12-18. *The code necessary to synchronize both List controls*

```
<mx:TabNavigator left="10" top="10" bottom="10" width="200"
    creationPolicy="all">

    <s:NavigatorContent label="Photo List" width="100%" height="100%">
        <s:List id="photoList" height="100%" width="100%"
            dataProvider="{photoXMLList}" labelField="@title"
            change="{thumbList.selectedIndex = photoList.selectedIndex}"/>
    </s:NavigatorContent>

    <s:NavigatorContent label="Thumbnails" width="100%" height="100%">
        <s:List id="thumbList" height="100%" width="100%"
            dataProvider="{photoXMLList}" itemRenderer="ThumbItemRenderer"
            change="{photoList.selectedIndex = thumbList.selectedIndex}"/>
    </s:NavigatorContent>

</mx:TabNavigator>
```

For a final adjustment, there's one last problem we need to remedy: we want the first image to load when the application is launched. We can handle this with the **HTTPService** component. Find the **photoService** component and append its attributes to include the **result** event emphasized in Example 12-19.

Example 12-19. *Setting the result event on the HTTPService component*

```
<s:HTTPService id="photoService" url="photos.xml" resultFormat="e4x"
    result="photoList.selectedIndex=0"/>
```

At this point, the **PhotoGallery** is functionally complete! But don't delete it, because we return to it later in Chapter 15 when we address the topic of styling using Cascading Style Sheets (CSS). As for now, Example 12-20 presents the final application code, and Figure 12-15 shows the result.

Example 12-20. *C:\Flex4Code\PhotoGallery\src\PhotoGallery.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    applicationComplete="photoService.send()">

    <s:layout><s:BasicLayout/></s:layout>

    <fx:Declarations>
        <s:HTTPService id="photoService" url="photos.xml" resultFormat="e4x"
            result="photoList.selectedIndex=0"/>
        <s:XMLListCollection id="photoXMLList"
            source="{photoService.lastResult.photo}"/>
    </fx:Declarations>
```



```

<mx:TabNavigator left="10" top="10" bottom="10" width="200"
    creationPolicy="all">

    <s:NavigatorContent label="Photo List" width="100%" height="100%">
        <s:List id="photoList" height="100%" width="100%"
            dataProvider="{photoXMLList}" labelField="@title"
            change="{thumbList.selectedIndex = photoList.selectedIndex}"/>
    </s:NavigatorContent>

    <s:NavigatorContent label="Thumbnails" width="100%" height="100%">
        <s:List id="thumbList" height="100%" width="100%"
            dataProvider="{photoXMLList}"
            itemRenderer="ThumbItemRenderer"
            change="{photoList.selectedIndex = thumbList.selectedIndex}"/>
    </s:NavigatorContent>

</mx:TabNavigator>

<mx:Image id="photoImage" source="{photoList.selectedItem.@image}"
    horizontalAlign="center" verticalAlign="middle"
    left="220" top="10" bottom="10" right="10"
    open="progressBar.visible = true"
    complete="progressBar.visible = false"/>

<mx:ProgressBar id="progressBar" source="{photoImage}"
    width="100" bottom="15" right="30"
    visible="false"/>

</s:Application>

```

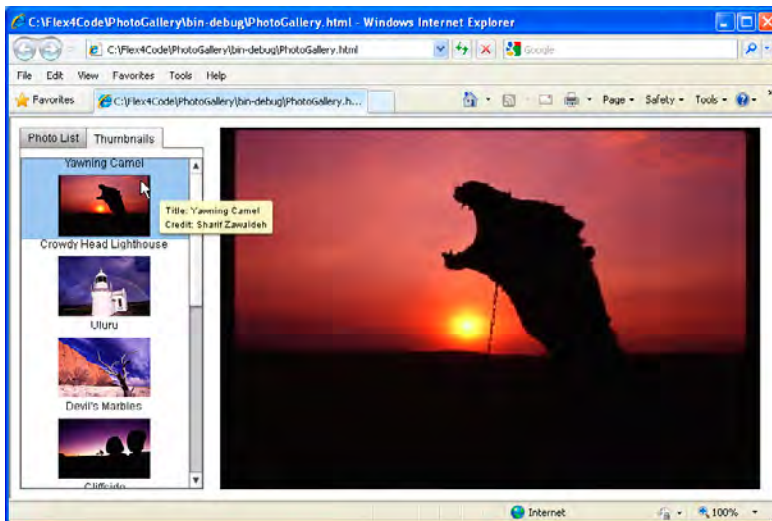


Figure 12-15. The PhotoGallery application in Thumbnail view

Now that you've learned about navigator containers and created the **PhotoGallery**, you're encouraged to modify the application using different navigation controls and containers. For instance, could you scroll the thumbnails horizontally along the bottom of the application? Can you think of some other compelling layouts that might improve the existing arrangement?

Summary

In this chapter, you learned how to control visibility flow using the Flex navigator controls. Navigator controls give you some real authority over your screen real estate, and they also help organize different parts of an application, such as forms and categorized content. Navigators also provide a means of hiding components that might not be needed and that would otherwise clutter the screen.

You used navigation components to create customized views for the **PhotoGallery** application, providing both a simple **List** of labels as well as a customized thumbnail view. You also learned how to load images using an XML file that can be loaded either from a directory relative to the application or from the Web. You even learned how to monitor loading progress. This chapter also reaffirmed some important concepts, such as data binding, layout construction, loading external data, and creating a custom **ItemRenderer**.

Although navigation controls have undeniable potential, there's yet another way to make a fluid interface: view states. Chapter 13 introduces you to view states, and it'll give you yet another "iron in the fire," as the saying goes.

WORKING WITH VIEW STATES

“How’s that shopping cart coming?”

...

...

“...how ’bout now?”

—Chris Giese

View states can add another level of dynamics to your UI. States allow you to present different layout arrangements to support specific conditions. For instance, an application that recognizes different permissions between user accounts might provide a read-only state for some users while enabling an editing state for managers and administrators.

States provide a clean method of grouping modest UI changes into organized, understandable chunks. In other words, a state is a collection of changes to properties, styles, or behaviors of your UI components.

Of course, a developer could accomplish the same results using a series of functions to impose a batch of changes to the UI, but the MXML approach used by Flex is both easier to read and quicker to write—and with Flash Builder, you can always use Design mode to help you get started.

Scenarios for States

Think about the **YahooSearch** application you built in Chapter 11. In that example, the person using your utility sees an empty results list and the search field when she loads the application. Although there’s nothing mechanically wrong with that, it doesn’t make sense to show the results list until there are actual results to view. You could set the results list to invisible when the application is started, then set it back to visible when the search returns results; however, you might want a few more changes to occur in the UI when results appear.

Consider this: wouldn’t it be nice if the search field was the center of attention when the application opens, and then, once someone submits her search, the search field moves to the top, making room for the list of results?

IN THIS CHAPTER

- Scenarios for States
- Managing States in Design Mode
- Making a Login/Registration Form
- Applying States to the Search Application
- Summary

(See Figures 13-1 and 13-2.) Not only would this scenario be cool and dynamic, but it would help guide people through the application, showing them only what is necessary at a given time.

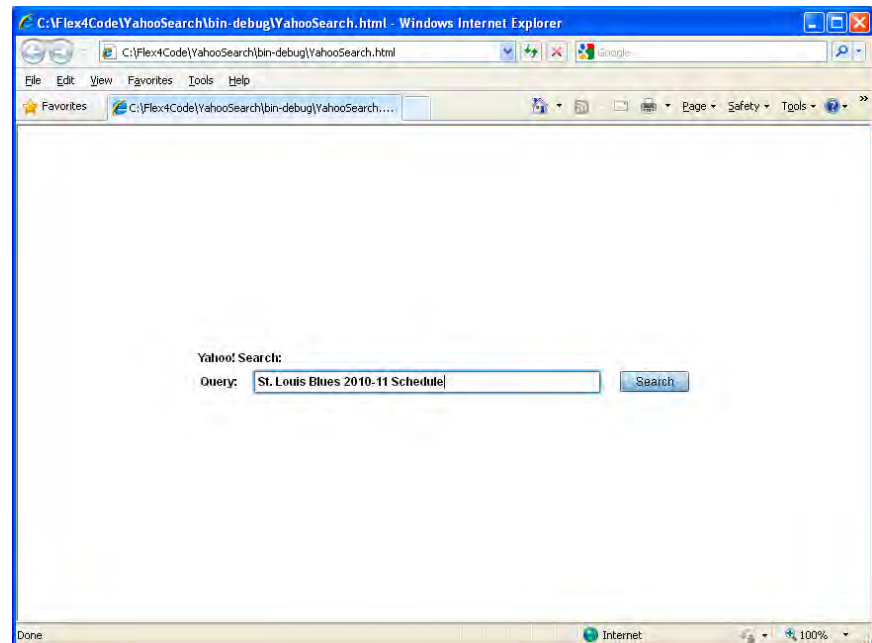


Figure 13-1. The Yahoo! search application in its initial state, showing only the search field

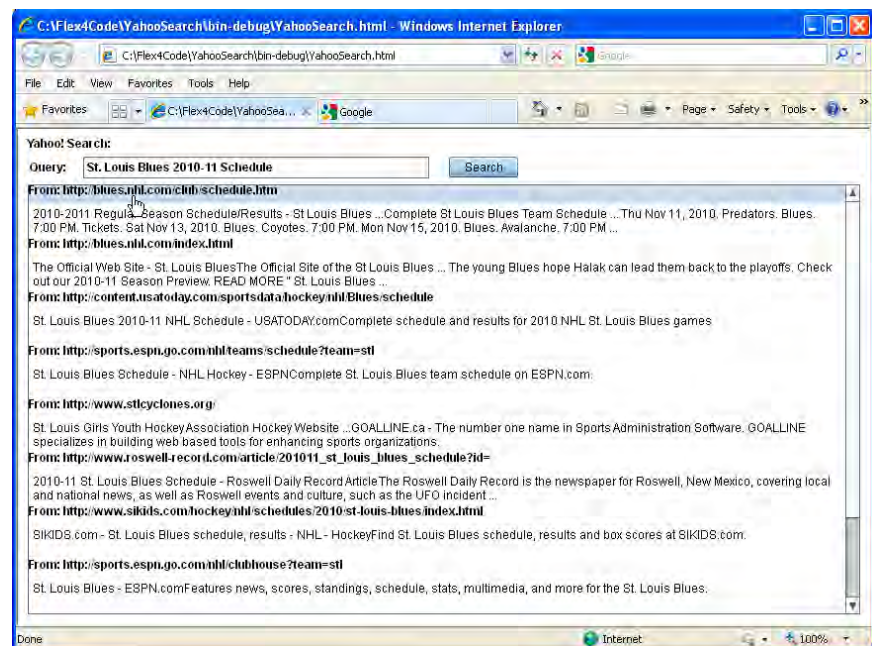


Figure 13-2. The search application in its running state, with the search field moved to the top and the list of results added

We begin working with states in Design mode by creating a simple demo. Afterward, we show you how to integrate state functionality into real project scenarios.

To get started, let's make a simple application that has two states, **stageLeft** and **stageRight**, that we'll use to move a **Button** back and forth between both sides of the screen.

Managing States in Design Mode

First, create a fresh project called **StageRight**, switch to Design mode, delete the **Application** container's **height** and **width** properties, and give the application a **BasicLayout**. This demo uses a single **Button**, so drag a **Button** control into the application and drop it somewhere near the lower-left corner.

Creating a New State

Now we'll use the States pane to create a new state. If the States pane isn't visible, you can show it by selecting Window→States. In the States pane, you should see **State1**, the default state. All applications are considered to have at least one state, which is their default, or *base state*. You can create a new state based upon the base state by clicking the New State button in the States pane, as shown in Figure 13-3.

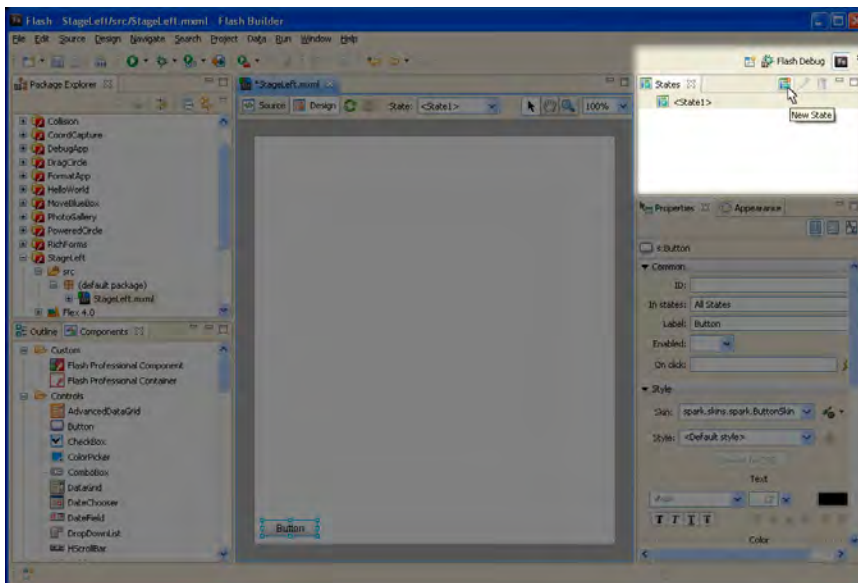


Figure 13-3. Creating a new state in the States pane

States Versus Navigator Containers

States can provide more flexibility than navigators.

First of all, states can build upon one another, and navigators cannot. In other words, the next state can be based upon the previous state, and you can add components progressively as you need them. However, for navigator containers, each view is independent of the others.

Moreover, components can be reused between the states. A control present in one state can continue to exist in another state, perhaps with a modified location or properties. With navigator containers, you would have to re-create the component if you wanted it to appear in two different views.

This isn't to say that view states are always the right choice. As we saw in the previous chapter, navigators have their uses. For instance, they're great for when the application needs to be categorized into separate parts. Also, navigation controls provide easy switching between content areas that have extensive differences, which would become complicated to implement with states.



Figure 13-4. The New State dialog box



Figure 13-5. The Edit State Properties dialog box

Clicking the New State button brings up a dialog box asking for information about the new state, as shown in Figure 13-4. You can give the state any name you want, but as always, it's best to be descriptive. For this example, call the state **stageRight** because all this state will do is move the **Button** to the right.

Editing State Properties

To keep our code clean and readable, let's edit the generically named **State1** so it's in keeping with our project. Depending on your OS, either right-click or Control-click the **State1** item and select Edit. When the Edit State Properties dialog opens, change the name of the base state to **stageLeft** and check the "Start state" option, as shown in Figure 13-5.

Modifying Layouts for States

Now that we have our states ready, we need to make them different from one another. To do this, we'll change the **Button** control's positioning properties for each state so that the **Button** is in the lower-left corner while the application recognizes **stageLeft**, and then it moves to the lower-right corner when the application recognizes **stageRight**. To change component properties for a state, first select the state you want to modify in Design mode's State selection box, as shown in Figure 13-6.

NOTE

Design mode gives you two ways to toggle between states while editing: the States pane and the State selection box.

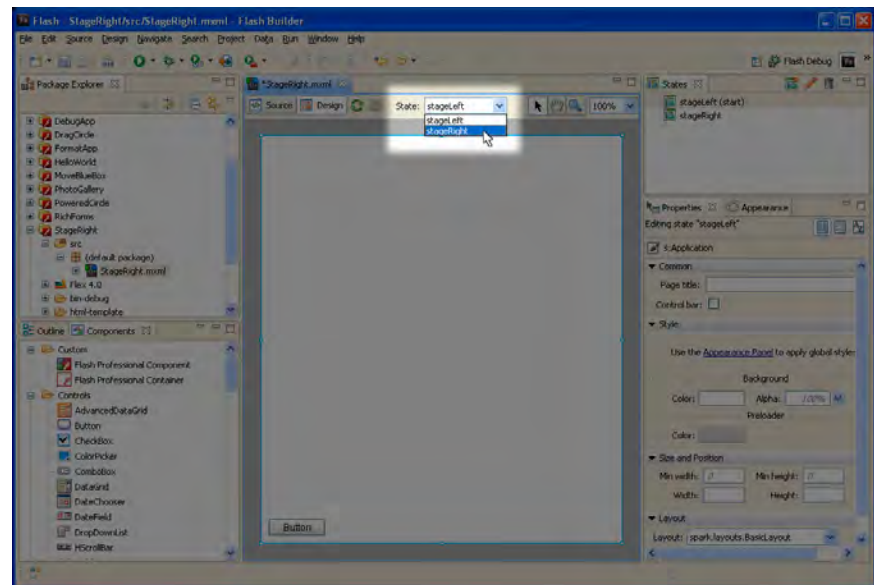


Figure 13-6. Selecting stageLeft in the State selection box

Starting with **stageLeft**, give the **Button** control **bottom** and **left** constraint values of 50 and 50. Next, return to the State selection box and choose **stageRight** (alternatively, you can merely select **stageRight** in the States pane). With Design mode now recognizing edits for **stageRight**, give the button **bottom** and **right** constraint values of 50 and 50.

While you're at it, notice something about the Properties pane; it's informing you of the state you're currently editing for. Just below the Properties tab in the top-left corner of the Properties pane, you should see:

Editing state "stageRight"

This provides you with a helpful reminder if you have closed your States pane (Figure 13-7).

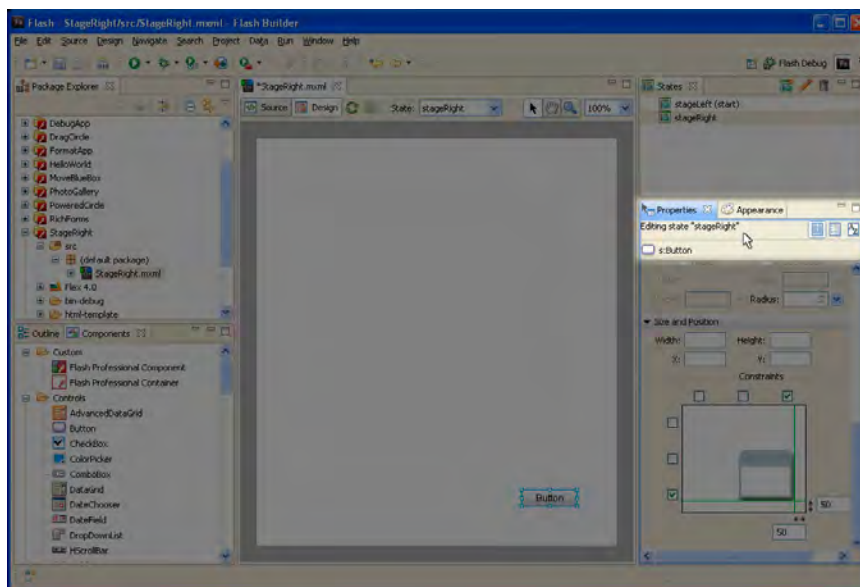


Figure 13-7. Confirming the editing state in the Properties pane

At this point, you can use either the States pane or the State selection box to toggle between **stageLeft** and **stageRight** while in Design mode. However, one more detail remains before you can run this demo and change states on the fly, and that's an event trigger to change the state.

Changing the Current State with Script

For the last step, you need to jump into Source mode so we can write a function to test for and change the application's **currentState** property. Once you jump into Source mode, you should see code similar to Example 13-1, though likely less formatted.

WARNING

When modifying states in Design mode, pay close attention to the state you're editing. If you don't, you might unintentionally modify properties of a component for a different state, which could create a lot of frustration as you try to identify and correct the error, especially if you've proceeded deep into the edit history.

Example 13-1. States code created by Design mode

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  currentState="stageLeft">

  <s:layout>
    <s:BasicLayout/>
  </s:layout>

  <s:states>
    <s:State name="stageLeft"/>
    <s:State name="stageRight"/>
  </s:states>

  <s:Button label="Button"
    left.stageLeft="50"
    bottom.stageLeft="50"
    bottom.stageRight="50"
    right.stageRight="50"/>

</s:Application>

```

Example 13-1 contains mostly unfamiliar code. In the attribute properties of the **Application** container, notice the **currentState="stageLeft"** assignment. This was set when we renamed the base state and set it to be the starting state. This is also the property we will test and reassign using script. Notice also the two MXML **State** declarations; their main purpose is making the states available to the application.

It's the **Button** control's properties that are doing all the work. When multiple states are available to an application, you can prepend inline property assignments with a state name using dot notation (such as **bottom.stageRight**), which tells the compiler you want to apply that property to one particular state. So, after glancing over the **Button** properties created by Design mode, you can see the two sets of constraint assignments for **stageLeft** and **stageRight**.

Now that we've explained what you're seeing, you should create the Script/CDATA block shown in Example 13-2 in the usual place, just above the layout declaration.

Example 13-2. A function to handle state change

```

<fx:Script>
  <![CDATA[
    private function changeState():void{
      if(this.currentState == "stageLeft"){
        this.currentState = "stageRight";
      }else{
        this.currentState = "stageLeft";
      }
    }
  ]]>
</fx:Script>

```


The purpose of the `changeState()` function is to switch the application's `currentState` to its counterpart. The `this` keyword represents the instance of the class calling the function; here, `this` refers to the application itself. The function uses an `If..Else` block, and within the condition, the *equality operator* (`==`) tests the value of the `currentState` property. We built the function relative to `stageLeft`, but you could easily invert it to work off of `stageRight` instead.

To call the function, add an inline `click` listener to the `Button` that calls the `changeState()` function, as shown in Example 13-3.

Example 13-3. Calling the `changeState()` function using inline script

```
<s:Button label="Button" click="changeState()"
    bottom.stageLeft="50"
    left.stageLeft="50"
    right.stageRight="50"
    bottom.stageRight="50"/>
```

Go ahead and run the project to see it at work. Beyond discussing the process of making and editing states, this application is pointless, but hopefully you have a working answer to the question “What are states?” and you’re more comfortable creating and managing states in a Flex application. In the next section, we consider a familiar application of states and how they might be applied to a login/registration form.

Making a Login/Registration Form

Consider the example of a username/password login form, common to many web applications. The typical experience is this: you visit a website that asks for a username and password, and if you don’t have an account with the site, you have the option of registering. To register, you usually have to load a new page to get to the registration form. On the other hand, wouldn’t it be nice if you could stay on the same page and simply trigger the registration fields as needed? That’s exactly what we demonstrate in the next example.

The Base State

To get started, use either Design mode or Source mode to put together the following application in its base state. In a new project, call the application `LoginForm`. We provide the code in Example 13-4. See Figure 13-8 for the visual.

Example 13-4. Base state code for the Login/Registration form

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">
```

Figure 13-8. The login form in its base state

```

<fx:Script>
  <![CDATA[
    private function toggleRegisterLogin():void{
      // handle state change
    }
  ]]>
</fx:Script>

<s:layout>
  <s:VerticalLayout horizontalAlign="center" verticalAlign="middle"/>
</s:layout>

<s:Panel id="loginPanel" title="Returning Users Sign In">

  <s:layout>
    <s:VerticalLayout horizontalAlign="right"
      paddingLeft="5" paddingRight="5"
      paddingTop="5" paddingBottom="5"/>
  </s:layout>

  <mx:Form height="100%" width="100%">

    <!-- we'll put the full name component here -->

    <mx:FormItem label="Username:">
      <s:TextInput id="usernameTI"/>
    </mx:FormItem>
    <mx:FormItem label="Password:">
      <s:TextInput id="passwordTI" displayAsPassword="true"/>
    </mx:FormItem>

    <!-- we'll put the confirm password component here -->

  </mx:Form>

  <s:Button id="submitButton" label="Sign In"/>

  <s:controlBarLayout>
    <s:HorizontalLayout horizontalAlign="right"/>
  </s:controlBarLayout>

  <s:controlBarContent>
    <mx:LinkButton id="registerLinkButton"
      label="Don't have an account yet?"
      click="toggleRegisterLogin()"/>
  </s:controlBarContent>

</s:Panel>

</s:Application>

```

This is pretty standard stuff. However, at the bottom of the code, you have the `controlBarLayout`, `controlBarContent`, and `LinkButton` classes. Essentially, the `controlBar` classes allow us to finish a `Panel` with alternative components that take the same styling as the `Panel`'s title area. The `LinkButton` component provides `label` and `click` properties that allow a line of text to behave like a `Button`.

Since we wanted the **LinkButton** to stay to the right, just below the Sign In button, we applied a **HorizontalLayout** with `horizontalAlign="right"` to the **controlBarLayout**. Next, we added the **controlBarContent** container and nested the **LinkButton**.

This takes care of the base state for the login form, but now we need to create our view states and apply their functionality.

Adding View States in Source Mode

Our login/registration form is going to need two states: **login** and **registration**. You could easily jump into Design mode, create the **registration** state, and then create the **login** state and set it as the start state, but you can also handle this in Source mode with five lines of code! Assuming you're in Source mode, add the MXML in Example 13-5 just below the Script/CDATA block.

Example 13-5. Adding state declarations in Source mode

```
<s:states>
  <s:State name="login"/>
  <s:State name="registration"/>
</s:states>
```

Those four lines get you started by enabling the two states for the application. Next, set the default state by assigning the **Application** container's **currentState** property, as shown in Example 13-6. Code completion should help you select the correct state.

Example 13-6. Assigning the *currentState* property in Source mode

```
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  currentState="login">
```

That's it. With five lines of code, you just added two states and set the base state.

NOTE

Once you've added state declarations to a class—in this case, your **Application**—they become available to code completion.

Including and Excluding Components from States

Besides the state property assignments you saw previously in the **StageRight** application, you also can include and exclude components from states using the **includeIn** and **excludeFrom** properties, which are available to all visual components.

For the login/registration form, you'll need the **registration** state to *add* two form fields, one for the user's full name, and another for password reentry, the strategy generally used to validate a new user's password. You'll add the Full Name **FormItem** and the Confirm Password **FormItem** as the respective top and bottom items in the **Form**. As you're creating the **includeIn** property assignments, code completion should suggest the view states you created in the previous step.

Example 13-7 shows the modifications you'll need to make to the form; both the Full Name and the Confirm Password **FormItem** components need to be added. Note the use of the **includeIn** property to define which state these components should recognize.

Example 13-7. *Adding the Full Name and Confirm Password FormItem components and defining their state assignments*

```
<mx:Form height="100%" width="100%">

    <mx:FormItem label="Full Name:" includeIn="registration">
        <s:TextInput id="fullNameTI"/>
    </mx:FormItem>

    <mx:FormItem label="Username:">
        <s:TextInput id="usernameTI"/>
    </mx:FormItem>

    <mx:FormItem label="Password:">
        <s:TextInput id="passwordTI" displayAsPassword="true"/>
    </mx:FormItem>

    <mx:FormItem label="Confirm Password:" includeIn="registration">
        <s:TextInput id="confirmPassTI" displayAsPassword="true" />
    </mx:FormItem>

</mx:Form>
```

Rather than including the Full Name and Confirm Password components in the **registration** state, we could have *excluded* them from the **login** state. To do this, we would have used the **excludeFrom** property to eliminate either item from the **login** state, which would have required something like Example 13-8.

Example 13-8. *The excludeFrom="login" option, which would exclude the component from the login state, yet make it available to the registration state*

```
<mx:FormItem label="Full Name:" excludeFrom="login">
    <s:TextInput id="fullNameTI"/>
</mx:FormItem>
```

Linking Component Properties to States

The next step in this example requires assigning state-specific properties to a handful of components in the login/registration form.

Recall from the short and sweet first example that you assign state-specific properties to a component using inline property assignments that consist of the property name, then a dot followed by a state name, followed by the actual value assignment. Add the property assignment in Example 13-9 to the **Panel** container so that its title will change when the application takes the **registration** state.

Example 13-9. *Creating a state-specific property for the Panel's title*

```
<s:Panel id="loginPanel"
    title="Returning Users Sign In"
    title.registration="New User Registration">
```

After typing **title.**, code completion should allow you to attach a state assignment from the application's nested states, and then carry on with the rest of the value assignment. Easy enough, right? Next, provide a **label** for the application's **Button** that's unique to the **registration** state, as shown in Example 13-10.

Example 13-10. *Creating a state-specific property for the submitButton label*

```
<s:Button id="submitButton"
    label="Sign In"
    label.registration="Register"/>
```

Finally, as shown in Example 13-11, fix the **LinkButton** so that its **label** will be more appropriate for the **registration** state.

Example 13-11. *Creating a state-specific property for the registerLinkButton label*

```
<mx:LinkButton id="registerLinkButton"
    label="Don't have an account yet?"
    label.registration="Already have an account with us?"
    click="toggleRegisterLogin()"/>
```

Changing the Current State with Script

The last step, which applies an **ActionScript** function to perform the state change, concludes the exercise, and it's identical to the approach used to change states in the previous example. Merely complete the hollow function block with the **If...Else** statement shown in Example 13-12.

Example 13-12. *Adding an **ActionScript** function to handle the state change*

```
private function toggleRegisterLogin():void{
    if(this.currentState=="login"){
        this.currentState="registration";
    }else{
        this.currentState="login";
    }
}
```

Once you run the application, use the **LinkButton** to jump between the **login** and **registration** states. The new **registration** state should resemble Figure 13-9.

NOTE

In the next chapter we show you how to make state changes more glamorous, by using transitions and effects.

Figure 13-9. *The login form in its registration state*

Applying States to the Search Application

You should be getting comfortable working with state code in Flex, but we still have one more exercise: applying states to the **YahooSearch** application.

Restoring the Application

Return to the search application you created in Chapter 11. For the sake of starting in the same place, Example 13-13 presents the **YahooSearch** code we assume you're returning to.

WARNING

Don't forget, the **YahooSearch** application uses an **ItemRenderer** we saved in the package `C:\Flex4Code\com\learning-flex4\renderers\`.

Example 13-13. *The YahooSearch application as we left it in Chapter 11*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:yahoo="http://www.yahoo.com/astra/2006/mxml"
  defaultButton="{searchButton}">

  <fx:Declarations>
    <yahoo:SearchService id="searchService"
      applicationId="YahooSearch"
      query="{queryTI.text}"/>
  </fx:Declarations>

  <s:VGroup left="10" right="10" top="10" bottom="10">
    <s:Label text="Yahoo! Search:" fontWeight="bold"/>

    <s:HGroup>
      <mx:FormItem label="Query:" fontWeight="bold">
        <s:TextInput id="queryTI" width="350"/>
      </mx:FormItem>
      <mx:FormItem>
        <s:Button id="searchButton" label="Search"
          click="searchService.send()"/>
      </mx:FormItem>
    </s:HGroup>

    <s:List id="resultsList" width="100%" height="100%"
      itemRenderer="learningflex4.renderers.SearchItemRenderer">
      <s:layout>
        <s:VerticalLayout/>
      </s:layout>
      <s:dataProvider>
        <s:ArrayCollection source="{searchService.lastResult}"/>
      </s:dataProvider>
    </s:List>

  </s:VGroup>
</s:Application>
```

Revising the Search Application

The application code provided in Example 13-13 loads search results through an **ItemRenderer**. It looked OK while running, but it left something to be desired at startup. See for yourself in Figure 13-10. Basically, the interactive controls are jammed in the upper-left corner of the screen, and there's just something undesirable about that empty **List**. In this exercise, we use states to fix these shortcomings.

Here's what we're going for in the revised application:

- We want the start state to show the search field only, front and center; the results list should not be visible at startup.
- Once we conduct a search, we want the search field to reposition at the top of the window and the results list to appear below it, occupying the rest of the window.

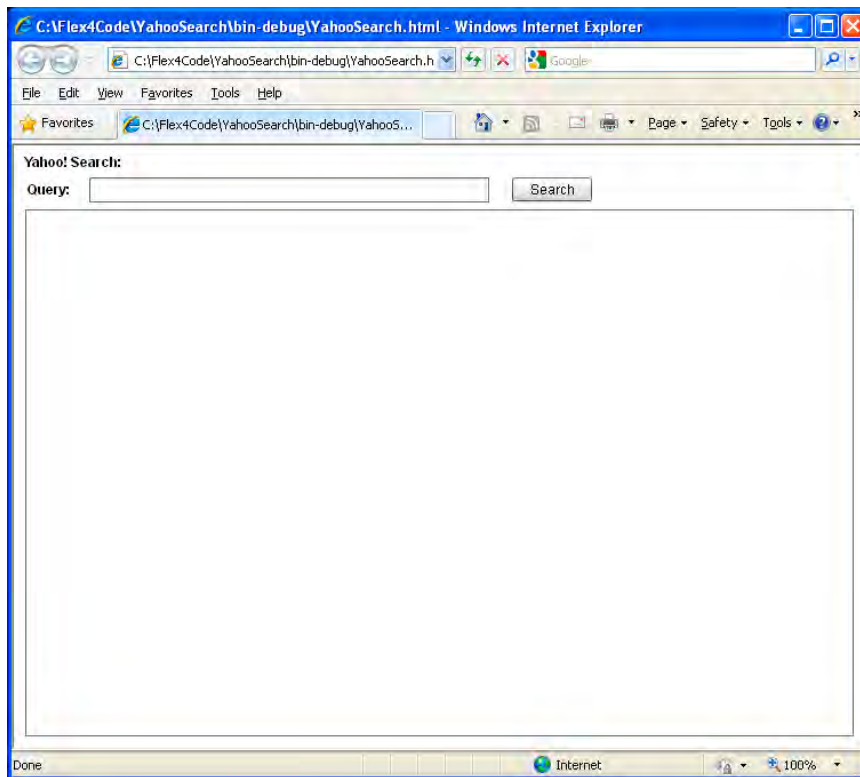


Figure 13-10. The search utility with its naked, pre-use appearance

To make this happen, we need to create two states: **initial** and **running**. We'll set the application's start state to **initial**, use the **excludeFrom** property to exclude the **List** from the **initial** state, and finally, add a handful of properties to move the search field between states. There are a few other details we'll address in the moment. It might sound like a lot, but it's really not. You'll see.

Adding States and State-Specific Properties

In the previous example, we learned to begin view state developments by declaring the states and assigning their name values. So, using your preferred method, create two states—**initial** and **running**—and for the **Application**'s **currentState** property, assign **initial** as the startup state. Those actions will add the code emphasized in Example 13-14 to the top of the application.

Example 13-14. Adding states to the *YahooSearch* application

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:yahoo="http://www.yahoo.com/astrea/2006/mxml"
  defaultButton="{searchButton}"
  currentState="initial">
```

```

<s:states>
  <s:State name="initial"/>
  <s:State name="running"/>
</s:states>

```

Find `resultsList` and prepare to make some changes. To start, delete the nested layout declaration. Next, set the control's `excludeFrom` property to `initial`, as shown in Example 13-15, so `resultsList` won't be available to the `initial` state.

Example 13-15. *Excluding `resultsList` from the initial state, eliminating it from startup*

```

<s:List id="resultsList" width="100%" height="100%"
  itemRenderer="learningflex4.renderers.SearchItemRenderer"
  excludeFrom="initial">

  <s:dataProvider>
    <s:ArrayCollection source="{searchService.lastResult}"/>
  </s:dataProvider>

</s:List>

```

For the next step, assign some state-specific properties to the `VGroup` container. The attribute modifications presented in Example 13-16 will create movement of the search field from the center of the application to the upper left corner when the application's `currentState` is changed.

Example 13-16. *State-specific properties of the `VGroup` that will cause the search field to move from the middle center to the top-left corner of the application*

```

<s:VGroup horizontalCenter.initial="0"
  verticalCenter.initial="0"
  height.running="100%"
  width.running="100%"
  left.running="10"
  right.running="10"
  top.running="10"
  bottom.running="10">

```

Last, notice how the Button's `click` event is handled: it's inline within the MXML block. We'll use the `searchButton`'s `click` event to trigger the state change as well as call the search service's `send()` method, so we should handle the task with an appropriate function. As such, add the Script/CDATA block shown in Example 13-17 near the top of the application code.

Example 13-17. *Handling the state change and the `HTTPService send()` method in a named function*

```

<fx:Script>
  <![CDATA[
    private function searchButtonClick():void{
      if(!(this.currentState == "running")){
        this.currentState="running";
      }
      searchService.send();
    }
  ]]>
</fx:Script>

```


There's something new here. Note the condition in the `if` block, particularly the use of the exclamation mark (!):

```
if(!(this.currentState == "running")){
```

The exclamation mark (!) is called the *inequality operator*. We only want to change the application's `currentState` property from `initial` to `running` at the time of the very first search. Afterward, we'll leave the application in the `running` state. The inequality operator tells the application, "If the `currentState` does not equal `running`, change the `currentState` to `running`."

Last but not least, change the `Button` control's `click` event to call the new function, as shown in Example 13-18.

Example 13-18. *Modifying the button's click event to call the named function*

```
<mx:FormItem>
    <s:Button id="searchButton" label="Search"
        click="searchButtonClick()"/>
</mx:FormItem>
```

That's a wrap. Go ahead and run the application and observe how the state additions improve the dynamics, aesthetics, and flow of the `YahooSearch` application (Figures 13-11 and 13-12).

NOTE

Another, perhaps clearer way of writing the conditional expression for Example 13-17 would be as follows:

```
if(this.currentState != "running"){
    this.currentState="running";
```

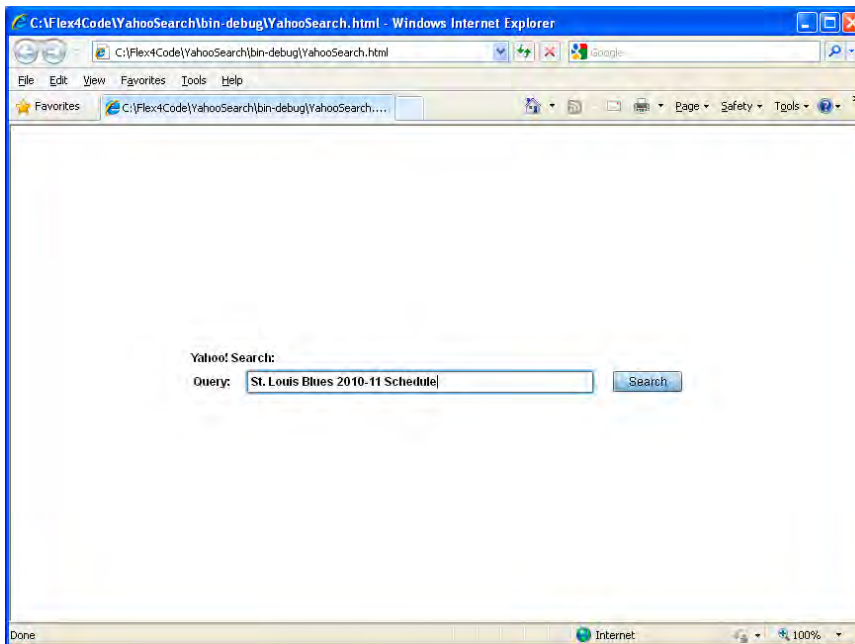


Figure 13-11. *The search utility in its initial State*

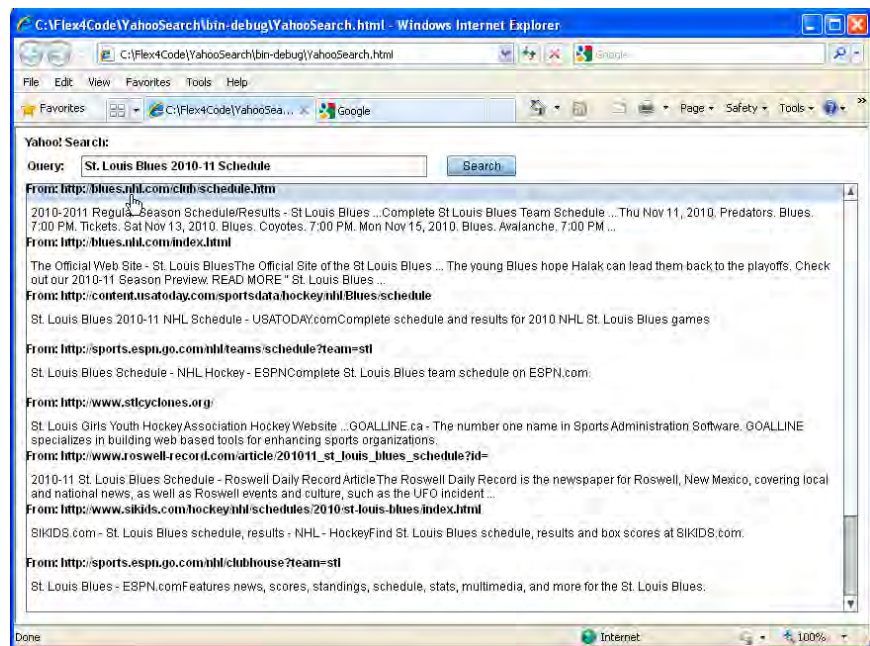


Figure 13-12. The search utility in its running State

Summary

By now you should be comfortable creating and editing states using either Design mode or Source mode. You should be able to create new states and use them to organize subtle layout modifications and case-specific component arrangements. You know that you can include or exclude components from states, and you've created state-specific property assignments in component MXML using inline declarations and dot notation. You know how to approach state development, and best of all, you've applied states to a couple of realistic scenarios.

In the next chapter, we give our applications some Flex appeal by learning about effects, behaviors, transitions, and filters—the stuff that really turns heads.

APPLYING EFFECTS, TRANSITIONS, AND FILTERS

*“The Royal Hawaiian was wonderful...it was the ideal life.
After twenty-four hours of it we were bored stiff.”*

—Edward L. Beach,
Run Silent, Run Deep

IN THIS CHAPTER

Effects

Transitions

Filters

Summary

Certain things may come to mind when you think about applying effects to your applications. Wining and dining. Sugar and spice. Whistles and bells. Sliced bread. OK, maybe not sliced bread, but you get the point: these are some of our favorite things.

So where do effects fit into the big picture? Like the previously mentioned niceties, effects should come in small doses and accentuate a larger experience. How would pumpkin pie taste if someone dumped an entire bottle of nutmeg into the filling? And how do you feel about emergency sirens at 3 a.m.? Not good. But with the right amount of nutmeg, the pie tastes great. Similarly, the sound of an air horn at a hockey game gets people excited. Think of effects in terms of these contexts. Less is more.

How then can effects benefit your applications? Generally effects, transitions, and filters—called *behaviors* when referenced altogether—should be used as attention-getters, drawing attention to the right thing at the right time. Rollover effects on menus and buttons provide the classic example of purposeful animation. However, behaviors can also be fun and expressive, and maybe there’s a place for that in your applications.

This chapter introduces you to Flex behaviors and shows you how to apply them. First, you’ll experiment with some simple animation examples, and then you’ll sprinkle some transitions into the **YahooSearch** application. Finally, you’ll create a test bed application to experiment with filter properties.

Effects

To return to programming contexts, effects animate changing component properties, size, position, or overall appearance characteristics. For example, a static **alpha** value of 0.5 assigned to a graphic object is not an example of an effect, but the animated change of an object's **alpha** value from 1 to 0 is an effect. In fact, it's called **fade**, and we'll use it in an example shortly.

Before we discuss the categories of and specific effects available to you, let's demonstrate how to declare and play an effect animation.

Effects Basics

Although observed visually, effects are nonvisual objects. As such, they should be placed in the Declarations section of your application code. Example 14-1 demonstrates how to declare a **fade** effect; notice it has an **id** property, which we'll need later when we want to play the effect's animation.

NOTE

*Effect animations can be triggered by obvious events such as **click**, **mouseover**, **resize**, **creationComplete**, etc., but as we'll see later, state changes work too. With state changes, though, the event is implied by the changing state, and it's called a transition.*

Example 14-1. Effects should be added in the Declarations section

```
<fx:Declarations>

    <s:Fade id="fadeOut" alphaFrom="1" alphaTo="0" duration="1000"/>

</fx:Declarations>
```

Once the effect is declared, you'll need an event or a function procedure to set it in motion. Next, in Example 14-2, we use a **Button** to call **fadeOut** on itself via its **click** event.

Example 14-2. Calling an effect and passing it an Array of ids, even though there's only one item in the Array

```
<s:Button id="button" label="Play Effect" left="50" bottom="50"
    click="fadeOut.play([button])"/>
```

Specifically, the button's **click** event is calling the **play()** method on **fadeOut**, and it's passing into the **play()** method an **Array**, recognizable by the presence of brackets ([]), with a single item—itsself—as represented by its own **id** property.

If you create a quick application to test that code, you'll find it makes your **Button** fade out elegantly.

Now that you know how to declare an instance of an effect class and call it on a component, let's discuss the categories of events available to you.

Effects Categories

Effects are conceptually divided into the following categories: property effects, transform effects, 3D effects, filters, pixel-shading effects, masks, and blends. Here's an itemized breakdown of these effects categories:

NOTE

*This example shows how to pass a component's **id** into an effect as an item in an **Array**. You can optionally hardcode an effect's target(s) within the effect's MXML declaration, but that's not all; you can also make loose effects that target any component meeting certain conditions, such as resizing or moving. Later in the chapter we cover these different tactics available for assigning effect targets.*

Property effects

These behaviors animate changing property values such as opacity and transparency, component size values, colors, and other attribute assignments. The **Animate**, **Fade**, **Resize**, and **AnimateColor** effects belong in this category.

Transform effects

Behaviors in this category animate positional movement, rotations, and scaling, all in 2D against the x-axis and y-axis. This category includes the **Move**, **Rotate**, and **Scale** effects.

3D effects

For the most part, 3D effects are just like the transform effects, only they allow for animations about the x-axis, y-axis, and the z-axis. The z-axis represents the plane of the screen display, with **z=0** being directly upon the screen, positive **z** values falling into the screen, and negative **z** values moving above the screen. These effects include **Move3D**, **Rotate3D**, and **Scale3D**.

Filters

Filter effects include **DropShadow**, **GlowFilter**, **BlurFilter**, **BevelFilter**, **ConvolutionFilter**, and some others. These filters apply a static change to some properties of a visible element, meaning they are not animated.

Pixel-shading effects

This category of effects animates before and after renderings of images and components. The available effects include **Crossfade** and **Wipe**, but you can also create custom pixel-shading effects. The pixel-shading effects are most commonly applied to images.

Masks

Masks can take one of three formats, as declared by the constants **CLIP**, **ALPHA**, and **LUMINOSITY**. Basically, a mask imposes the **Fill** shape of one graphic upon the **Fill** shape of another graphic.

Blends

Blend modes operate as you might think; they take overlapping graphics and force them to intersect one another in the color space. By default, the last graphic added to the display list appears above graphics added before it, but blend modes allow you to bypass this behavior, forcing graphics to merge together.

NOTE

The potential to manipulate images using pixel-shading, masks, and blends runs quite deep, and we chose not to cover these categories of effects. If you want a primer on these topics, though, check out the Flex 4 Cookbook, by Joshua Noble et al. (<http://oreilly.com/catalog/9780596805623/>).

Applying Spark Effects

There are times when you'll want to drive multiple effects with the same event. Sometimes you'll want them to occur back-to-back, but other times you may want them to occur simultaneously. Fortunately, Flex allows you to specify such combinations in your MXML using either **Sequence** or **Parallel** tags, respectively.

You also have options for how you can assign effects to their target components and elements. As we saw in the first demo, one option is to pass an **Array** of targets to the effect's **play()** method, but we can also specify the target or targets directly within the effect's MXML declaration.

In the examples that follow, we demonstrate how to apply Spark effects to an FXG graphic. To better understand FXG graphics, see the sidebar below titled “FXG Graphics.” If you want to use the same graphic we use in the examples, just download it from the following link and save it in your project's *src* folder: <http://www.learningflex4.com/fxg/CoolTriangle.fxg>

FXG Graphics

The ability to consume FXG graphics is something new to Flex 4. Simply, **FXG graphics** are graphic object classes that are declared using XML syntax rules. You can create and export FXG graphics using either Adobe's Illustrator or Photoshop products, and then plug those graphics directly into your Flex applications.

The point of FXG graphics is to simplify their creation using design software and open a door for graphic designers to create skins for custom components. Plus, if you've ever created a graphic programmatically, you know that's not an ideal approach. Now that you know what FXG graphics are and why they are useful, the next question is, how can you create some for yourself?

If you have Adobe Illustrator, that's the best platform for creating vector graphics and saving them as FXG files. Assuming you have a graphic you want to export out of Illustrator, follow

File→Save As, set the file type to FXG, give it a name, and then save it to an appropriate directory.

With the FXG file ready for Flex, simply copy it into your **src** directory or a special package, and treat it like any other class you can declare and assign an **id**. The name of your FXG file will become the name of your graphic class; we'll show you an example in a moment.

At the time of this writing, 7jigen.net provides a free online utility you can use to create FXG graphics code. It has a few goofy quirks in its present incarnation, but it's definitely a great idea for a RIA. Check it out at <http://fxgeditor.7jigen.net/>.

Figure 14-1 demonstrates graphic creation using 7jigen's editor. Once the graphic is ready, clicking the “FXG” button presents the code for your graphic, as shown in Figure 14-2.

(continued)

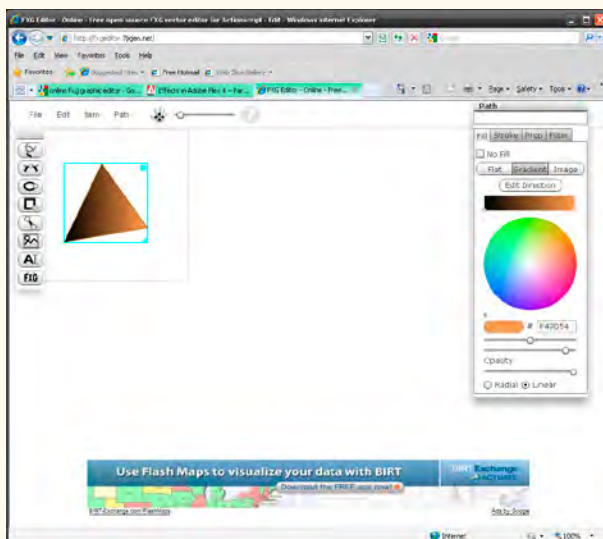


Figure 14-1. Creating an FXG graphic object using the online editor at 7jigen.net

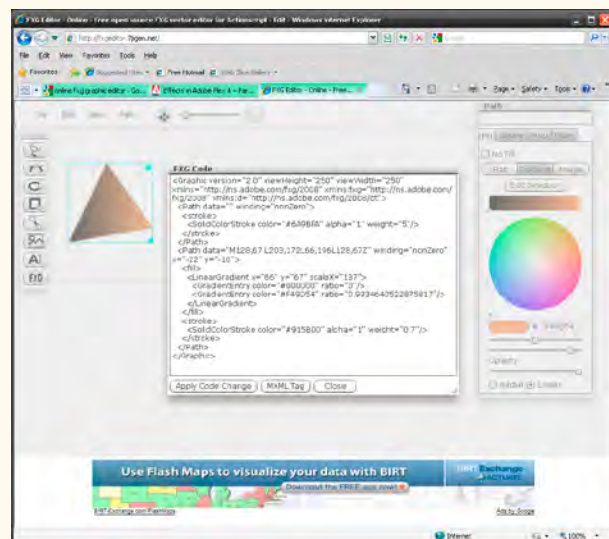


Figure 14-2. Getting FXG code from the 7jigen editor

FXG Graphics (continued)

The 7jigen editor produced code very similar to the following (we cleaned it up a little bit) for the triangle you see in Figure 14-2:

```
<Graphic version="2.0"
  xmlns="http://ns.adobe.com/fxg/2008"
  xmlns:fxg="http://ns.adobe.com/fxg/2008"
  xmlns:d="http://ns.adobe.com/fxg/2008/dt"
  viewHeight="250" viewWidth="250">

  <Path data="M128,67 L203,172L66,196L128,67Z"
    winding="nonZero" x="-22" y="-10">
    <fill>
      <LinearGradient x="66" y="67"
        scaleX="137">
        <GradientEntry color="#000000"
          ratio="0"/>
        <GradientEntry color="#F49D54"
          ratio="1"/>
      </LinearGradient>
    </fill>

    <stroke>
      <SolidColorStroke
        color="#915B00"
        alpha="1"
        weight="0.7"/>
    </stroke>

  </Path>

</Graphic>
```

If you're following this approach, your next step would be to create an empty FXG file in your Flex project's *src* folder (or any folder below the *src* directory) and then copy the FXG code into that file. To create a blank FXG file in Flash Builder, simply open File→New→File, browse to the correct project and *src* location, and then provide a fitting name for the graphic (Figure 14-3). Make sure to include the *.fxg* extension!

Once you select Finish, Flash Builder will open a new editor for your FXG file. At this point, just copy the code created by the 7jigen editor and save. That's it. The graphic is ready to

use. To reference the code in your project MXML, start typing the filename you provided for the graphic. Code completion should help you finish the declaration and automatically add the correct namespace manifest and class instance, like so:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:local="*"
  height="100%" width="100%">
```

```
<!-- other code between these -->
```

```
<local:CoolGraphic />
```

Of course, it's up to you to give your component an *id* attribute and some placement properties; otherwise, the compiler will use defaults.



Figure 14-3. Creating a new FXG file in Flash Builder

NOTE

To follow along with these examples, you can download the FXG graphic we used from the companion website: <http://www.learningflex4.com/fxg/CoolTriangle.fxg>.

Effects in sequence

If you want to play two or more effects back-to-back, you'll need to nest those effects in a **Sequence** block. Our first example demonstrated a fade-out effect—which is a property effect—on a **Button**, but let's change that code so a fade-out *and* a fade-in play back-to-back, in sequence. This time, we'll use an FXG graphic as the target. See Example 14-3.

Example 14-3. *Playing a fade-out followed by a fade-in by running the animations in sequence*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:local="*"
  height="100%" width="100%">

  <fx:Declarations>

    <s:Sequence id="sequenceFade">
      <s:Fade id="fadeOut" alphaFrom="1" alphaTo="0" duration="1000"/>
      <s:Fade id="fadeIn" alphaFrom="0" alphaTo="1" duration="1000"/>
    </s:Sequence>

  </fx:Declarations>

  <local:CoolTriangle id="coolTriangle"
    horizontalCenter="0" verticalCenter="0"/>

  <s:Button id="button" label="Play Effect"
    left="50" bottom="50"
    click="sequenceFade.play([coolTriangle])"/>

</s:Application>
```

This code does nearly the same thing as the example we opened the chapter with, only instead of calling a specific effect instance using its **id**, we triggered a sequence of effects by referencing the unique **id** of the **Sequence: sequenceFade**.

Effects of a **Sequence** block will occur in order of their declaration within the block, so in this case, we have a fading out followed by a fading in (Figure 14-4).

Effects in parallel

You also have the option of playing two or more effects simultaneously. To do that, nest some effect declarations inside a **Parallel** block.

This combination, shown in Example 14-4, will fade out the graphic *while* rotating it about its y-axis using the **Rotate3D** effect.

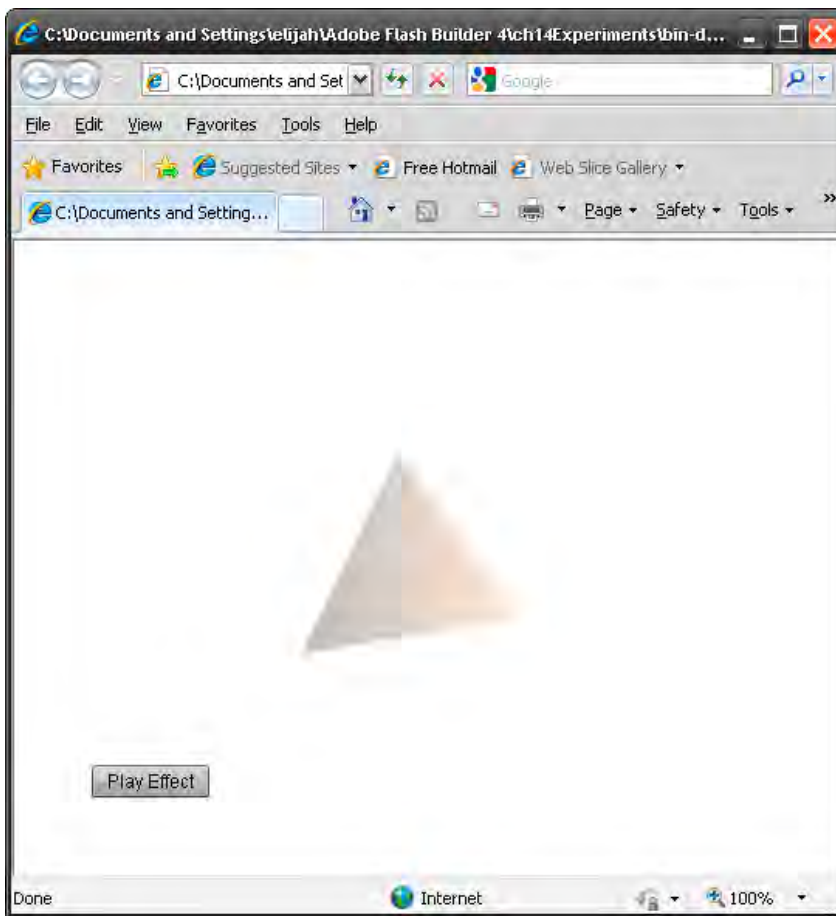


Figure 14-4. Catching a fade effect in the act of fading

Example 14-4. Playing a fade-out simultaneously with a 3D rotation by running the animations in parallel

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:local="*"
  height="100%" width="100%">

  <fx:Declarations>

    <s:Parallel id="parallelTwist">
      <s:Fade alphaFrom="1" alphaTo="0" duration="1000"/>
      <s:Rotate3D angleYFrom="0" angleYTo="360" duration="1000"
        autoCenterTransform="true"/>
    </s:Parallel>

  </fx:Declarations>
```

```

<local:CoolTriangle id="coolTriangle"
    horizontalCenter="0" verticalCenter="0"/>

<s:Button id="button" label="Play Effect"
    left="50" bottom="50"
    click="parallelTwist.play([coolTriangle])"/>

</s:Application>

```

Besides using a new effect type, **Rotate3D**, the main difference between this example and the previous example is the word **Parallel**. However, the resulting difference is pretty obvious. To see for yourself, modify your code and give it a spin, literally (see Figure 14-5).

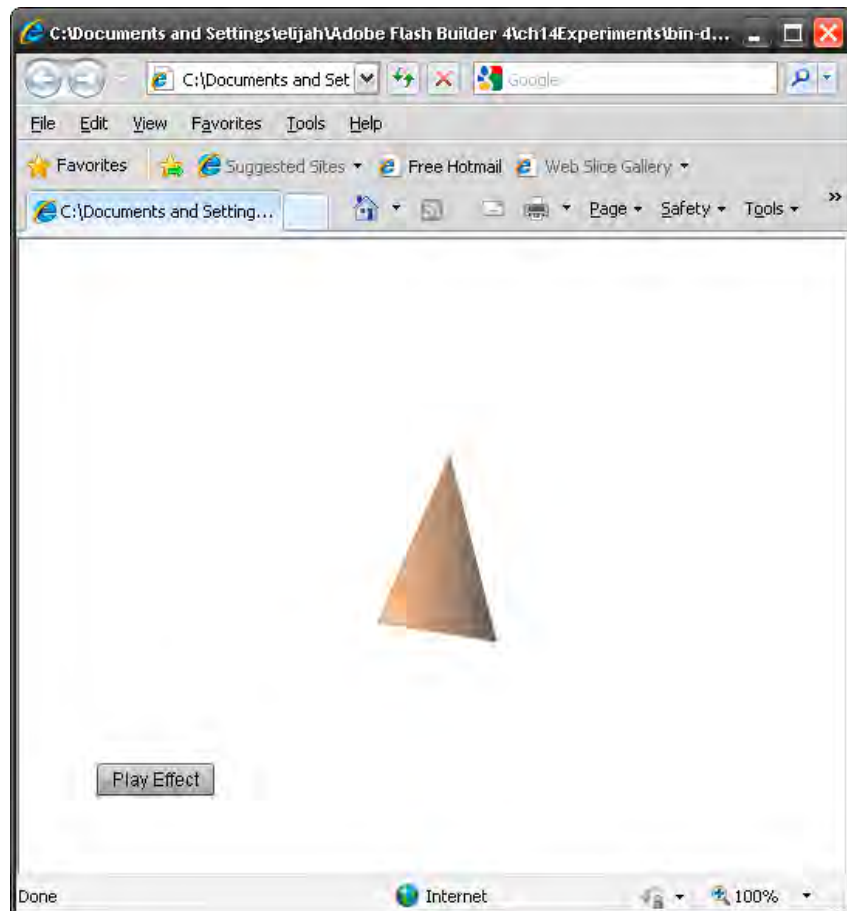


Figure 14-5. The fading, rotating triangle graphic

Let's take a moment to break down that 3D effect; we expand it here to better expose its parts:

```
<s:Rotate3D
  angleYFrom="0"
  angleYTo="360"
  duration="1000"
  autoCenterTransform="true"/>
```

angleYFrom

This property represents the starting angle of the rotation, where **0** represents the object at its original layout position. It accepts values 0 through 360 as degrees. If you start an object rotation at a value greater than 0, it will jump to your specified value and begin rotating from there.

angleYTo

This property specifies the ending angle of a rotation. Unlike **angleYFrom**, **angleYTo** can take positive and negative numbers, and it can take values that exceed 360 in both directions. If **angleYTo** is less than **angleYFrom**, the result will be a counterclockwise rotation. Positive values exceeding 360 will cause repeat rotations, and the same concept applies to counterclockwise rotations.

duration

The **duration** is the length of time in milliseconds an effect will require to play to completion. So far, we've used long durations to help visualize the effect; you should experiment with some long and short durations to get a feel for their impact on an animation.

autoCenterTransform

This is an important property. By default, transform effects are applied relative to the top-left corner of their target display object. Frequently, though, you're going to want an animation to occur relative to the true center of your object. For these situations, set **autoCenterTransform="true"** to bypass the default behavior. To visualize the difference, remove the **autoCenterTransform** property from the previous example and run it again.

Combining sequenced and parallel effects

You can arrange effects back-to-back with a **Sequence** block, and you can run multiple effects simultaneously with a **Parallel** block. You can also combine these two systems. Example 14-5 contains some code that plays a fading sequence while rotating the triangle—this time rotating the graphic about its x-axis.

Example 14-5. Combining a fade-in and fade-out in sequence with a 3D rotation running parallel to the sequence

```
<s:Parallel id="parallelFadeAndTwist">

    <s:Sequence>
        <s:Fade alphaFrom="1" alphaTo="0" duration="1000"/>
        <s:Fade alphaFrom="0" alphaTo="1" duration="1000"/>
    </s:Sequence>

    <s:Rotate3D angleXFrom="0" angleXTo="360" duration="2000"
        autoCenterTransform="true"/>

</s:Parallel>
```

To achieve this arrangement of effects, the **Sequence** block is nested inside the **Parallel** block, and the **duration** values for the two fade effects are each half the value of the rotation effect's **duration**. See Figure 14-6 to see the effects in action.

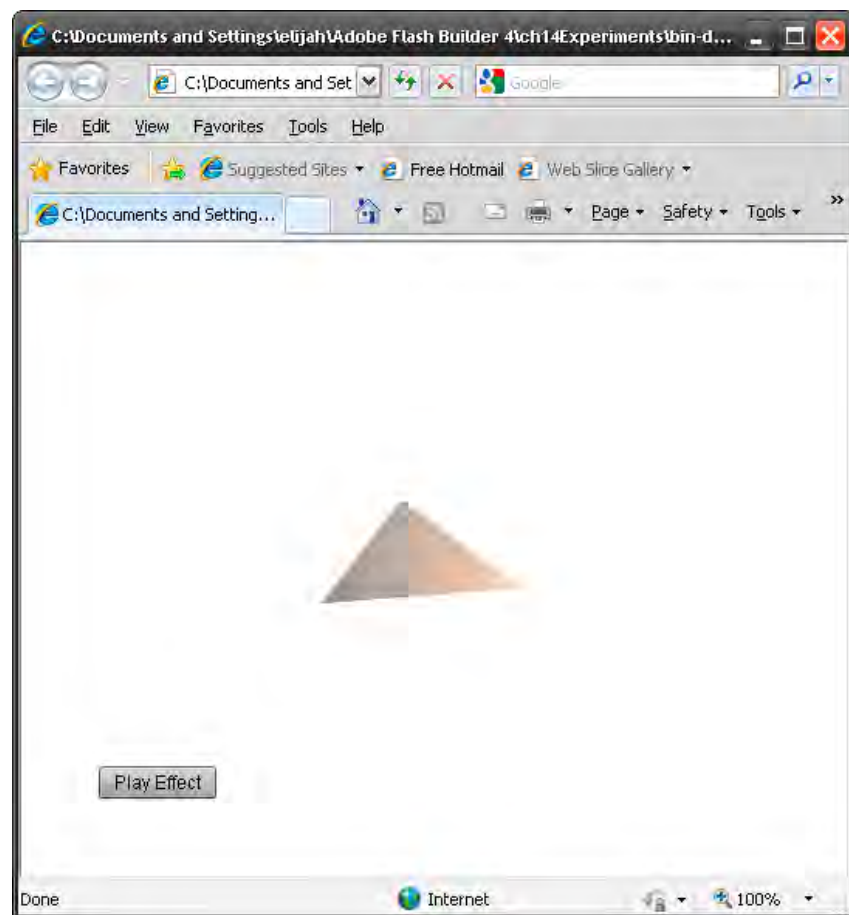


Figure 14-6. Combining sequence and parallel effects

Passed-in targets

So far, the effects examples we've shown have passed an **Array** of targets (even though we've only worked with a single target) into the effect instance for processing and animation. Here is the code structure we've been using:

```
effectID.play([graphicID])
```

If we wanted to render the effect on multiple targets, we would modify the ActionScript passing the target **Array** like so:

```
effectID.play([graphicID, componentID])
```

Or, to put it into the context of our previous examples, it might look like this:

```
parallelFadeAndTwist.play([coolTriangle, button])
```

If you modify the **click** event of the **Button** for the most recent example and run it, you'll see both the triangle graphic and the **Button** fading and rotating. By now, though, you should be comfortable with this approach, so let's move to another option you might want to use from time to time.

Explicit targeting

Sometimes, you might want to render an effect on a specific object or objects, and that's it. When this is the case, you can declare an effect's targets inside its own MXML declaration. If we revised the effect from the first example of this chapter to use an explicit target, it would look like Example 14-6.

Example 14-6. *A fade-out effect with a fixed, explicit target*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  height="100%" width="100%">

  <fx:Declarations>

    <s:Fade id="fadeOut" alphaFrom="1" alphaTo="0" duration="1000"
      target="{button}"/>

  </fx:Declarations>

  <s:Button id="button" label="Play Effect" left="50" bottom="50"
    click="fadeOut.play()"/>

</s:Application>
```

Because the effect already has a target assignment, you don't have to pass in targets when you call its **play()** method. However, if you wanted to apply the effect to *both* the button and the graphic, your effect declaration would need a subtle modification, shown in Example 14-7.

Know When to Say "When"

It's easy to get carried away with effects. Remember, websites present information and applications serve a purpose, and too many effects can create obstacles for these goals.

A curious effect that might be neat upon its first occurrence can become annoying in repetition. Similarly, any long-duration effect sequence can be unwanted if someone is in a hurry; we're thinking about those animated site intros that usually get skipped.

By other interpretations, too many effects can create a feeling of slow responsiveness for your application, and that's something you should avoid. Of course, the flip side is that well-used effects can buy time while you're waiting for data to load, thus giving the opposite impression of something, rather than nothing, happening.

With these thoughts in mind, and as a general rule, use effects sparingly. To repeat the sentiments of the chapter intro, less is more.

Example 14-7. A fade-out effect with two explicit targets; note the difference between the `target` property in the previous example and the `targets` property in this example

```
<s:Fade id="fadeOut" alphaFrom="1" alphaTo="0" duration="1000"
    targets="{[button, coolTriangle]}" />
```

Did you notice the difference? The effect's `target` property takes a single object `id` in curly braces (`{}`); the effect's `targets` property (note the "s") can take multiple objects by `id`, but multiples need to be wrapped in brackets (`[]`).

Effects can also be primed and played using ActionScript. Example 14-8 shows an effect being applied to a single target using a scripted function.

Example 14-8. Assigning a target to an effect and calling the effect's animation using ActionScript

```
<fx:Script>
    <![CDATA[
        private function buttonClick():void{
            fadeOut.target = button;
            fadeOut.play();
        }
    ]]>
</fx:Script>

<fx:Declarations>
    <s:Fade id="fadeOut" alphaFrom="1" alphaTo="0" duration="1000" />
</fx:Declarations>

<s:Button id="button" label="Play Effect"
    left="50" bottom="50"
    click="buttonClick()" />
```

And Example 14-9 demonstrates how you would apply that effect to multiple targets using script.

Example 14-9. Assigning multiple targets to an effect and calling the effect's animation using ActionScript

```
<fx:Script>
    <![CDATA[
        private function buttonClick():void{
            fadeOut.targets = [button, coolTriangle];
            fadeOut.play();
        }
    ]]>
</fx:Script>

<fx:Declarations>
    <s:Fade id="fadeOut" alphaFrom="1" alphaTo="0" duration="1000" />
</fx:Declarations>

<s:Button id="button" label="Play Effect"
    left="50" bottom="50"
    click="buttonClick()" />
```

About Halo Effects

In Flex 3, effects were called using special *effect triggers* inherited by all visual components, and many component interactions have ready-made triggers. Here's an example of a simple **Fade** animation called from a Halo **Image** component (in fact, this is the same **Image** component from our **PhotoGallery** application):

```
<mx:Image id="photoImage"
  source="{photoList.selectedItem.@image}"
  left="220" top="10" bottom="10" right="10"
  horizontalAlign="center"
  open="progressBar.visible = true"
  complete="progressBar.visible = false"
  completeEffect="Fade"/>
```

Under the Halo approach, you can call effects with default properties easily using only their names—**Fade**, **Glow**, **Dissolve**, **Move**, **Resize**, **Rotate**, etc.

You can also declare effects from the Halo package, assign them specific properties, and call them using these effect triggers. Just remember to stay within the MX namespace:

```
<fx:Declarations>
  <mx:Fade id="fadeIn"
    alphaFrom="0" alphaTo="1" duration="1000"/>
</fx:Declarations>

<mx:Image id="photoImage"
  completeEffect="{fadeIn}"/>
```

Adobe recommends you use the Spark effects, but you should be aware of the Halo approach, as you're likely to find examples on the Web and in older Flex books. Fair warning: you'll get strange results if you mix these approaches. If things don't seem to be working and you're not getting errors, make sure your namespaces are in sync.

Transitions

By now you should be fairly comfortable applying effects to your display objects and using events to trigger their **play()** method. However, you have another option: you can use state changes as an opportunity to trigger effects. In Flex terminology, it's called a *transition*.

Transition Syntax

For the most part, if you know how to program effects, then you know how to program state change transitions. The only real differences are where you declare the effects in your application code. Here's a breakdown of the differences:

- Transition effects are nested inside an `<s:Transition/>` block, which is nested inside an `<s:transitions/>` block; they are *not* placed in the Declarations section. They are typically placed just below any **State** definitions you've created.

```
<s:transitions>

  <s:Transition>

    <!-- effect goes here -->

  </s:Transition>

</s:transitions>
```

- The states affected by a transition are explicitly declared as property attributes of the `<s:Transition/>` instance—specifically, **fromState** and **toState**.

```
<s:transitions>

  <s:Transition fromState="firstState" toState="secondState">

    <!-- effect goes here -->

  </s:Transition>

</s:transitions>
```

- You'll handle target assignments for transition effects within the effects' MXML declarations. If the transitions are animating changing property values, those values are supplied behind the scenes by state-property assignments.

```
<s:transitions>

  <s:Transition fromState="firstState" toState="secondState">

    <s:Parallel target="{mainGroup}" duration="1000">

      <!-- animated values come from state-property assignments -->
      <s:Move/>

      <s:Resize/>

    </s:Parallel>

  </s:Transition>

</s:transitions>
```

NOTE

There is an alternative to this last approach called target filtering, which we discuss toward the end of this section.

Hopefully this breakdown doesn't muddy the waters—particularly since we haven't discussed the **Move** and **Resize** effects yet. To really get a sense for how to work with state change transitions, let's reload the **YahooSearch** application and give it some fun transitional effects.

Making State Changes More Interesting

To make sure we're starting in the same place, let's see the **YahooSearch** application as we left it, with two states: **initial** and **running**.

In the application code in Example 14-10, notice that we assigned **id** properties to the **VGroup**, **Label**, and **HGroup** components. Definitely update your code to include these **id** assignments, because our transitions will need them as targets.

Example 14-10. *Returning to the YahooSearch application as we left it in Chapter 13*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:yahoo="http://www.yahoo.com/astrea/2006/mxml"
  defaultButton="{searchButton}"
  currentState="initial">

  <fx:Script>
    <![CDATA[
      private function searchButtonClick():void{
        if(!(this.currentState == "running")){
          this.currentState="running";
        }
        searchService.send();
      }
    ]]>
  </fx:Script>

  <s:states>
    <s:State name="initial"/>
    <s:State name="running"/>
  </s:states>

  <fx:Declarations>
    <yahoo:SearchService id="searchService"
      applicationId="YahooSearch"
      query="{queryTI.text}"/>
  </fx:Declarations>

  <s:VGroup id="mainGroup"
    horizontalCenter.initial="0"
    verticalCenter.initial="0"
    height.running="100%"
    width.running="100%"
```

```

        left.running="10"
        right.running="10"
        top.running="10"
        bottom.running="10">

<s:Label id="titleLabel" text="Yahoo! Search:" fontWeight="bold"/>

<s:HGroup id="queryGroup">
    <mx:FormItem label="Query:" fontWeight="bold">
        <s:TextInput id="queryTI" width="350"/>
    </mx:FormItem>
    <mx:FormItem>
        <s:Button id="searchButton" label="Search"
            click="searchButtonClick()"/>
    </mx:FormItem>
</s:HGroup>

<s:List id="resultsList" width="100%" height="100%"
    itemRenderer="learningflex4.renderers.SearchItemRenderer"
    excludeFrom="initial">
    <s:dataProvider>
        <s:ArrayCollection source="{searchService.lastResult}"/>
    </s:dataProvider>
</s:List>

</s:VGroup>

</s:Application>

```

Building the transitions block

In your application code, between the **states** block and the Declarations block, start a new **transitions** block and begin an empty **Transition**. Set the **fromState** and **toState** properties to point the **Transition** at your **initial** and **running** states, as shown in Example 14-11.

NOTE

You can optionally configure your **fromState** property like this:

```
fromState="**"
```

Using this arrangement, any state can serve as the **fromState** when transitioning into the specified **toState**.

Example 14-11. Adding a transitions block and a Transition below the states block in the YahooSearch application

```

<s:transitions>
    <s:Transition fromState="initial" toState="running">

        </s:Transition>
    </s:transitions>

```

Now we need an effect for the **Transition**. Let's start with those **Move** and **Resize** effects lurking back there in the muddy water.

The **YahooSearch** application, as we know, begins with a **TextInput** front and center; then, when someone creates a search, that input field moves to the top left, and the search results are loaded into a **List** control. In fact, if you double-check the application code, the **VGroup**, which we gave an **id** of **mainGroup**, goes from a horizontal and vertical center to a constraints-based positioning that consumes most of the screen. This conversion constitutes both a moving and resizing.

If we apply what you know about effects, we can handle both the changing position (moving) and component size (resizing, to accommodate the constraints layout) simultaneously by nesting the effects within a **Parallel** block. So, create a **Parallel** block with **mainGroup** as the target, and then nest both a **Move** and a **Resize** effect, as shown in Example 14-12. By the way, **Move** is a transform effect, and **Resize** is a property effect. The component properties that are bound to one state or the other represent the changing layout and property values that will be passed to the nested effects.

Example 14-12. Adding the *Move* and *Resize* effects in a *Parallel* block targeting *mainGroup*

```
<s:transitions>
  <s:Transition fromState="initial" toState="running">

    <s:Parallel target="{mainGroup}" duration="1000">
      <s:Move/>
      <s:Resize/>
    </s:Parallel>

  </s:Transition>
</s:transitions>
```

At this point, you should run the application to see how the **Transition** plays into the state change.

Assuming you ran it, you should have observed the **VGroup** sliding from the center to the top left, meanwhile stretching to accommodate its new size. Since we used a duration of 1000 milliseconds—one complete, highly observed second—everything should happen just slowly enough to be noticeable.

We're only about one-third finished with our **Transition**, though. Let's add another bit of pizzazz to the **Parallel** block. You've already seen how to construct a **Rotate3D** effect, so add the following **Rotate3D** effect in the **Parallel** block. Since the rotation should only apply to the **TextInput** and its **label**, specify **queryGroup** as the target of the effect. Since we discussed them earlier under the section "Effects in parallel" on page 290, you should recall the **Rotate3D** properties. Example 14-13 provides the update for your growing **Transition** definition.

Example 14-13. Adding a Rotate3D effect to the Parallel animation

```

<s:transitions>
  <s:Transition fromState="initial" toState="running">

    <s:Parallel target="{mainGroup}" duration="1000">
      <s:Move/>
      <s:Resize/>
      <s:Rotate3D target="{queryGroup}"
        angleYFrom="0" angleYTo="360"
        autoCenterTransform="true"/>
    </s:Parallel>

  </s:Transition>
</s:transitions>

```

Once again, run the application to get a feel for the change.

There's only one more detail left to address with our transition, so we're getting close to finishing it. When the **YahooSearch** application changes state, the **resultsList** component is added and the search results are loaded into its rows. We'll also handle this in our **Transition** using an **AddAction** effect.

The **AddAction** effect is one of a handful of effects called *action effects*, intended just for state transitions. It's also intended to be wrapped in a **Sequence** block.

Here's the thinking: you already defined state properties for the application, and as such, you know some components will move and resize, and at the end, one more will be added. However, so far you don't have a mechanism to control *when* objects are added (or removed) during a state change. Now you do.

The point of **AddAction** is to control when **resultsList** is added to the layout. This is particularly important because we want to apply a fading-in effect to the list control. And since we're adding **resultsList** *after* the **mainGroup** is moved and resized, not simultaneously, we need to wrap our entire effects batch inside a **Sequence** block. Within the **Sequence**, the **Parallel** effects should come first, followed by **AddAction**, which ensures **resultsList** is added, followed by a **Fade** targeting **resultsList**.

Example 14-14 provides the next batch of code, which should explain itself.

Example 14-14. Extending the Transition with a Sequence block in order to time the addition of resultsList to the application

```

<s:transitions>
  <s:Transition fromState="initial" toState="running">

    <s:Sequence>

      <s:Parallel target="{mainGroup}" duration="1000">
        <s:Move/>
        <s:Resize/>
        <s:Rotate3D target="{queryGroup}"
          angleYFrom="0" angleYTo="360"
          autoCenterTransform="true"/>
      </s:Parallel>

    </s:Sequence>

  </s:Transition>
</s:transitions>

```

...aaaaand Action!

It can be difficult to know the exact order transition effects will follow, so Adobe created *action effects* to give you sequential control over their unveiling. Because action effects are used to control the order of transitional operations, they need to be nested in a **Sequence** block.

Our chapter example demonstrates an **AddAction**, which specifies when a visual object is added to the display, but the other action effects are used similarly. Here's the complete list of action effects available to you:

- **AddAction**: Adds a visible element to the display
- **RemoveAction**: Removes a visible element from the display
- **SetAction**: Modifies a property value
- **CallAction**: Calls a function or method of a defined target object

```

<s:AddAction target="{resultsList}"/>
<s:Fade target="{resultsList}" duration="2000"/>
</s:Sequence>

</s:Transition>
</s:transitions>

```

Go ahead and run this to observe how all the events unfold (Figure 14-7).

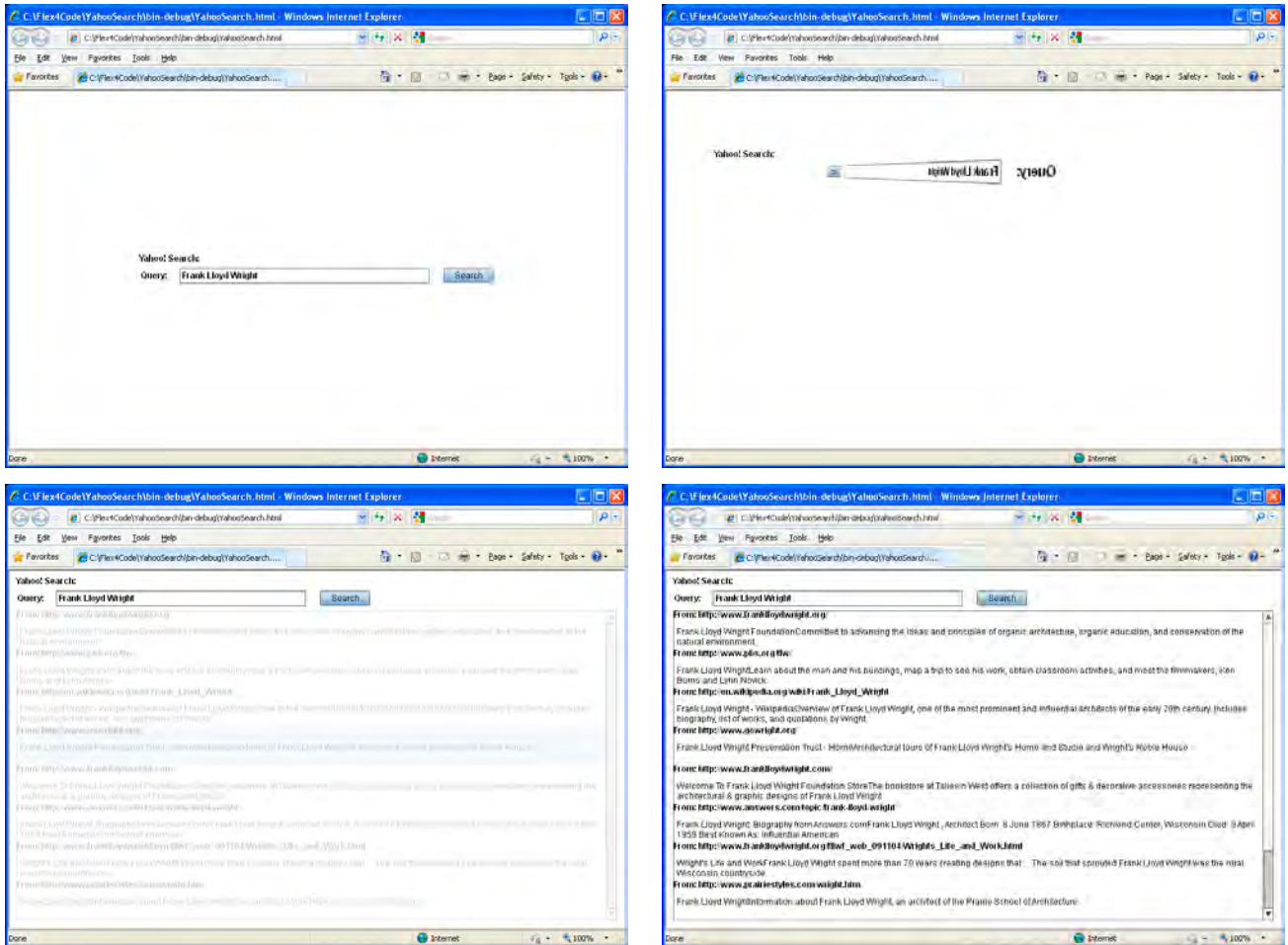


Figure 14-7. Combining sequence and parallel effects

Test the Waters

It's always a good idea to test your applications on different systems.

Although Flash is system agnostic, that doesn't mean it will run on every system just because it ran on yours. There's a menagerie of hardware/software combinations roaming around in the wild, and unless you experiment, you can't fully know which systems will play well with your applications.

Of course, there are always the obvious questions: "How does this work in Internet Explorer?" "How does it work on a Windows box using Google Chrome?" "How about Mac OS running Safari?" "Linux and Firefox?" No matter how many angles you test, for though, don't forget about the oldest system in your arsenal. Also, don't forget about the people stuck behind slow connections. Although you may not be able to modify your applications to accommodate every situation, you might be able to provide some warnings or cautions.

Frankly, this effects soup may be a bit extreme. To start, it's processor-intensive. The computer we used to compile the program gets "clippy" halfway through the transition, almost every time. If this series of effects occurred with every search, it would certainly become annoying. In its defense, though, it's a cool transition, and it occurs only once per session. So maybe it's OK? Of course, it's up to you to decide what is enough when it comes to your applications.

Target filtering (targets of opportunity)

In the previous example, we used explicit targeting to point each transition effect at its appropriate target. But there's another mechanism you can use to conditionally target transitioning elements based on how they're changing during a transition. It's called *target filtering*.

Here's how a target filter works: if during the course of a transition you wanted *any* resizing element to use the same resize event, you could declare the resize event with a filter condition, like in Example 14-15.

NOTE

A **Transition** declaration with the properties `fromState=""` and `toState=""` would apply to any state change.

Example 14-15. Using target filtering to apply a resize transition to any component that resizes during any state change

```
<s:transitions>
  <s:Transition fromState="" toState="">

    <s:Parallel targets="{[obj1, obj2, obj3, obj4]}">

      <s:Resize filter="resize"/>

    </s:Parallel>

  </s:Transition>
</s:transitions>
```

In this scenario, any component declared as a target of the **Parallel** effects (i.e., **obj1**, **obj2**, **obj3**, and **obj4**) could qualify for the filter. As such, any of these objects that resizes during any state change will receive the resizing animation, and all of those resizing components would be animated simultaneously.

Flex allows you to filter transitioning elements according to these conditions:

Add

Targets any element added during a transition

Remove

Targets any element removed during a transition

Show

Qualifies any element whose **visible** property changes from **false** to **true**

Hide

Qualifies any element whose **visible** property changes from **true** to **false**

Move

Targets any element with changing **x** or **y** properties

Resize

Targets any element with changing **height** or **width** properties

With this knowledge, let's revise the **YahooSearch** application to implement target filtering. Change your **Transition** as specified in Example 14-16.

Example 14-16. *Retooling the YahooSearch transition to apply target filters*

```
<s:transitions>
  <s:Transition fromState="initial" toState="running">
    <s:Sequence targets="[{mainGroup, titleLabel, queryGroup}]">
      <s:Parallel>
        <s:Move filter="move"/>
        <s:Resize filter="resize"/>
        <s:Rotate3D filter="move"
          angleYFrom="0" angleYTo="360"
          autoCenterTransform="true"/>
      </s:Parallel>
      <s:AddAction filter="add"/>
      <s:Fade filter="add" duration="2000"/>
    </s:Sequence>
  </s:Transition>
</s:transitions>
```

In this version, we're specifying all applicable targets up front, and each effect uses a filtering expression to qualify appropriate targets. We finish with essentially the same transition, but there is one difference (Figure 14-8). Can you spot it and explain why?

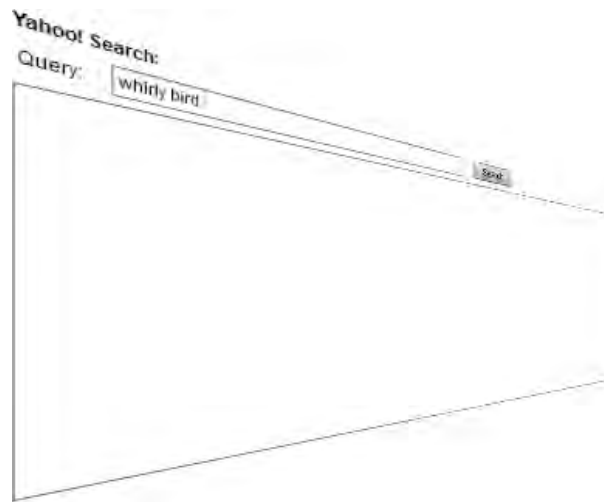


Figure 14-8. A slight difference after switching to target filtering

Filters

In previous examples you applied effects by declaring an effect, assigning a target or multiple targets as an **Array** to that effect, and then calling the effect's `play()` method. With filters, however, while you define their instances in your Declarations section, you bind them directly to your visual components as members of their **filters** property, which is also an **Array**. This is similar in concept, yet backward compared to the previous effects examples you've seen.

Example 14-17 contains a quick example to demonstrate setup and application of filter effects, and Figure 14-9 shows the result.

Example 14-17. Applying a `DropShadowFilter` effect to the FXG graphic

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  xmlns:local="*"
  height="100%" width="100%">

  <fx:Declarations>

    <s:DropShadowFilter id="dropShadow" distance="12"/>

  </fx:Declarations>

  <local:CoolTriangle id="coolTriangle" horizontalCenter="0" verticalCenter="0"
    filters="{[dropShadow]}" />

</s:Application>
```

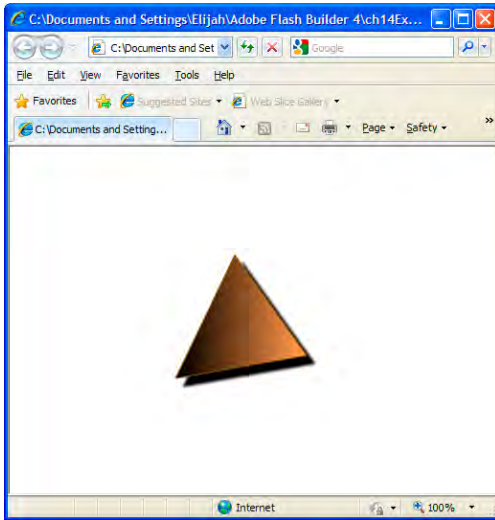



Figure 14-9. The triangle graphic with a `DropShadowFilter`

There shouldn't be any big surprises here. We declared a Spark `DropShadowFilter` in the **Declarations** section of the application code, making sure to give it an `id` property. We also defined a `distance` property of `12` to exaggerate the shadowy effect. Note that the default `distance` value would have worked. Finally, we apply the filter to our triangle graphic by assigning its `filters` property an `Array` containing a single filter object, `dropShadow`.

Now that you know how to define and attach a filter to a visual object, let's consider some of the types of filters available to you.

Types of Filters

If you want to quickly discover the Spark filter classes available to you, create a Script/CDATA block, start typing `import spark.filters.`, and you should be prompted with the full list of Spark filters (Figure 14-10). If you're feeling inquisitive, click on a filter to expose its documentation, and then read about it.

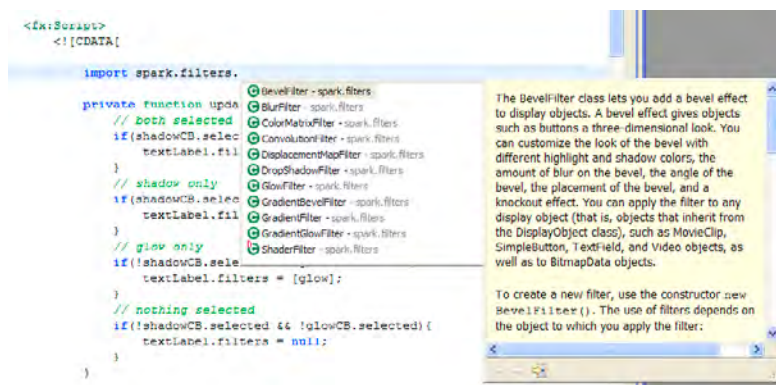


Figure 14-10. Exploring the Spark filter classes using code completion

NOTE

This example uses the FXG graphic we discussed in the sidebar “FXG Graphics” on page 288. If you want to keep things simple by using our graphic, just download it off the Web and save it in your project's `src` directory. You can get it here: <http://www.learningflex4.com/fxg/CoolTriangle.fxg>.

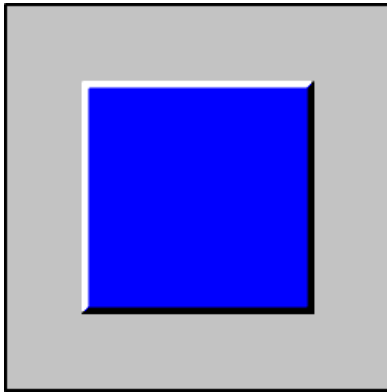


Figure 14-11. A `BevelFilter` applied to a `BorderContainer` object

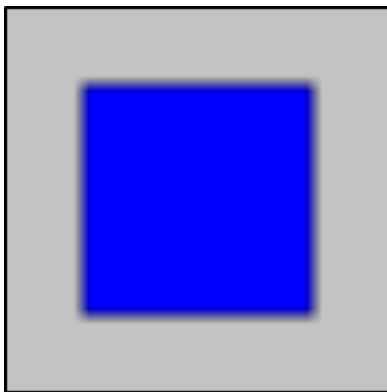


Figure 14-12. A `BlurFilter` applied to a `BorderContainer` object

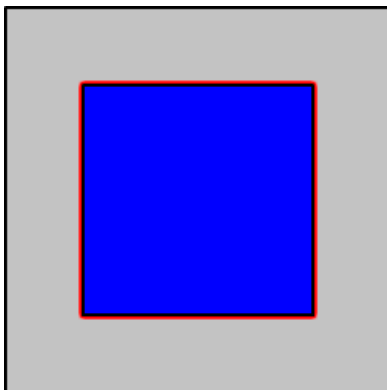


Figure 14-13. A `GlowFilter` applied to a `BorderContainer` object

Let's take a quick look at the Spark filters available to you:

`BevelFilter` and `GradientBevelFilter`

Bevels are commonly used to give a three-dimensional, chiseled look to a component (Figure 14-11). The `GradientBevelFilter` uses a gradient color across its bevel, improving the realism of the result.

`BlurFilter`

The `BlurFilter` provides an out-of-focus look to a component (Figure 14-12).

`DropShadowFilter`

This filter lets you easily add a drop shadow to any component, creating an illusion of depth, as if the component object is raised above the application (Figure 14-9).

`GlowFilter` and `GradientGlowFilter`

These two filters provide a glowing-edge effect on any component to which they are applied (Figure 14-13).

`ColorMatrixFilter`, `ConvolutionFilter`, and `DisplacementMapFilter`

These three filters are more complex than the others. To start, their use cases are more specialized and therefore occur less frequently. Second, many of their properties consume other objects as input. These are very powerful filters that can dramatically change the look of your components. The `DisplacementMapFilter` can even warp your entire application into a sphere!

The most widely used filters are probably the `DropShadow` filter and the `GlowFilter`, so those are the two filter classes we'll experiment with.

NOTE

*If you want to learn more about the `ColorMatrixFilter`, `ConvolutionFilter`, and `DisplacementMapFilter`, you can always review the *Flex 4* documentation, but if you anticipate doing a lot of graphic work, do yourself a favor and find a specialty book that discusses advanced raster graphics manipulation with ActionScript 3. The *ActionScript 3 Cookbook*, by Joey Lott et al. (<http://oreilly.com/catalog/9780596526955/>) includes several raster graphics manipulation strategies.*

Applying Filters

Let's build a demo application that uses Flex UI controls to set different effect properties. We're particularly interested in experimenting with filter effects, but we threw in some `Rotate3D` effects on the background container for the fun of it.

This time, let's see a screenshot of the application (Figure 14-14) before we look over the code.

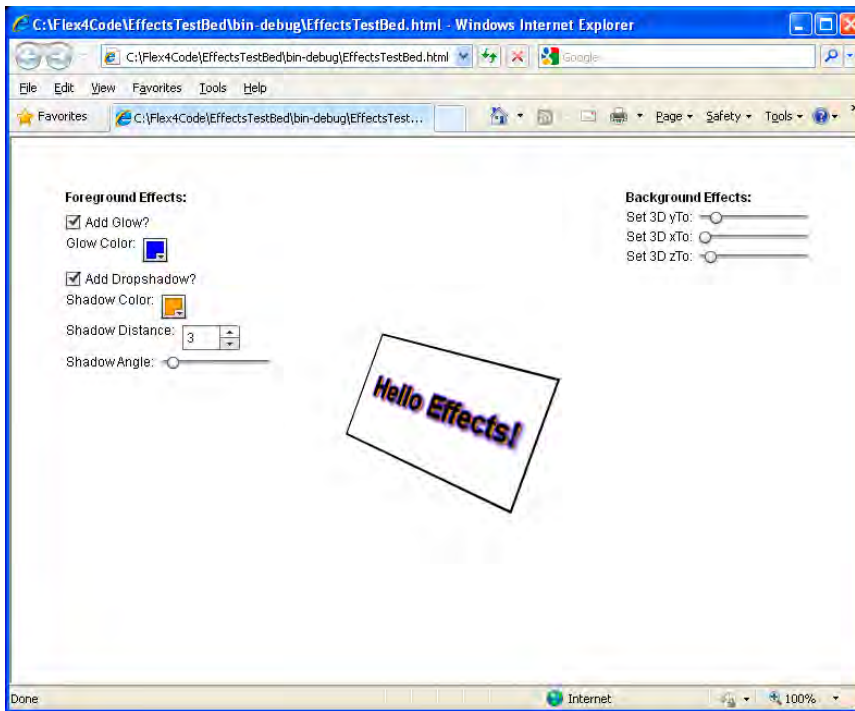


Figure 14-14. An interface for experimenting with effect properties

Many book examples and blog posts present applications like this as test beds for learning a technique, and it's about time we did the same. This application uses a **Label** control nested in a **BorderContainer** to experiment with different effect combinations. A number of UI controls are included, and their selectable values are bound to properties of the various effects. Specifically, we're working with the **GlowFilter** and the **DropShadowFilter**, which are attached to the **Label** control using two **CheckBoxes**: **glowCB** and **shadowCB**. There are a few details to be aware of as you look through the following code.

First of all, the **filters** property of the **Label** control is assigned using the function block, **updateFilters()**, which is called by the **change** event of either **CheckBox**. The **filters** property accepts an **Array** of filter objects. This is important and it introduces a hiccup: how do we maintain the label's **filters** array? Consider the problem: **glowCB** is already checked, and the **Label** control has a glow; next, if someone enables **shadowCB**, how should we make sure we maintain the glow while we add the drop shadow?

Although surely there are a number of ways to handle this problem, we solved it using a series of **if** conditions that involve the *inequality operator* (!) and the logical **AND operator** (&&) to look for every possible combination: Is only one checked? Are both checked? Is neither box checked? Let's consider the "glow only" condition in Example 14-18.

NOTE

The opposite of the logical AND operator (**&&**) is the OR operator (**||**).

Example 14-18. A compound conditional expression checking whether one of two `CheckBox` controls is enabled

```
if(!shadowCB.selected && glowCB.selected){
    textLabel.filters = [glow];
}
```

In this code, per the rules of the AND operator (**&&**), the expressions on either side of the **&&** symbols have to evaluate **true** for the full condition to be **true**. In the first half, **shadowCB.selected** would evaluate **true** if **shadowCB** were checked, but the inequality operator (**!**) causes the expression to evaluate as **true** only when the box is *unchecked*. The second half of the condition should be more obvious: **glowCB.selected** is **true** when **glowCB** is checked.

The combination of the slider controls and the **Rotate3D** effects might throw you a curveball, so we'll explain them briefly. The sliders themselves should be straightforward; because each slider has an **id**, its **value** property can be referenced using a data binding expression. Also, the sliders' **change** events are used to call the **play()** method of each effect. Their **minimum** and **maximum** values of 0 through 360 should make sense as the range of values accepted by **Rotate3D** for its **angleTo** property variants specific to each axis. However, the **Rotate3D** effect declarations are all missing their **angleFrom** properties. Why? Because we want the slider controls to impose instantaneous positional control, and if we used the **angleTo** properties, a rotation *animation* would occur each time an effect's **play()** method was called. For each of the sliders, the proper target is explicitly defined.

Example 14-19 provides the complete code for the effects-testing application, which we called **EffectsTestBed**.

Example 14-19. `C:\Flex4Code\EffectsTestBed\src\EffectsTestBed.mxml`

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="100%" width="100%">

    <fx:Script>
        <![CDATA[

            private function updateFilters():void{
                // both selected
                if(shadowCB.selected && glowCB.selected){
                    textLabel.filters = [shadow, glow];
                }
                // shadow only
                if(shadowCB.selected && !glowCB.selected){
                    textLabel.filters = [shadow];
                }
                // glow only
                if(!shadowCB.selected && glowCB.selected){
                    textLabel.filters = [glow];
                }
            }
        ]]>
    </fx:Script>
</s:Application>
```

```

        // nothing selected
        if(!shadowCB.selected && !glowCB.selected){
            textLabel.filters = null;
        }
    }
]]>
</fx:Script>

<fx:Declarations>

    <s:DropShadowFilter id="shadow"
        color="{shadowCP.selectedColor}"
        distance="{shadowDistNS.value}"
        angle="{shadowAngleSlider.value}"/>

    <s:GlowFilter id="glow"
        color="{glowCP.selectedColor}"
        strength="2"/>

    <s:Rotate3D id="textRotationY" target="{borderContainer}"
        angleYTo="{yToSlider.value}"
        autoCenterTransform="true"/>

    <s:Rotate3D id="textRotationX" target="{borderContainer}"
        angleXTo="{xToSlider.value}"
        autoCenterTransform="true"/>

    <s:Rotate3D id="textRotationZ" target="{borderContainer}"
        angleZTo="{zToSlider.value}"
        autoCenterTransform="true"/>

</fx:Declarations>

<s:BorderContainer id="borderContainer"
    borderColor="#000000" borderStyle="solid" borderWeight="2"
    horizontalCenter="0" verticalCenter="0">

    <s:Label id="textLabel" text="Hello Effects!"
        fontWeight="bold" fontSize="28"
        horizontalCenter="0" verticalCenter="0"
        paddingLeft="15" paddingRight="15"/>

</s:BorderContainer>

<s:VGroup top="50" left="50">
    <s:Label text="Foreground Effects:" fontWeight="bold"/>
    <s:CheckBox id="glowCB" label="Add Glow?"
        change="updateFilters()"/>
    <s:HGroup>
        <s:Label text="Glow Color: "/>
        <mx:ColorPicker id="glowCP"/>
    </s:HGroup>
    <s:CheckBox id="shadowCB" label="Add Dropshadow?"
        change="updateFilters()"/>
    <s:HGroup>
        <s:Label text="Shadow Color: "/>
        <mx:ColorPicker id="shadowCP"/>
    </s:HGroup>
</s:VGroup>

```

```

<s:HGroup>
  <s:Label text="Shadow Distance: "/>
  <s:NumericStepper id="shadowDistNS"
    minimum="1" maximum="20" value="2"/>
</s:HGroup>
<s:HGroup>
  <s:Label text="Shadow Angle: "/>
  <s:HSlider id="shadowAngleSlider"
    minimum="0" maximum="360" value="45"/>
</s:HGroup>
</s:VGroup>

<s:VGroup top="50" right="50">
  <s:Label text="Background Effects:" fontWeight="bold"/>
  <s:HGroup>
    <s:Label text="Set 3D yTo: "/>
    <s:HSlider id="yToSlider"
      minimum="0" maximum="360" value="0"
      change="textRotationY.play()"/>
  </s:HGroup>
  <s:HGroup>
    <s:Label text="Set 3D xTo: "/>
    <s:HSlider id="xToSlider"
      minimum="0" maximum="360" value="0"
      change="textRotationX.play()"/>
  </s:HGroup>
  <s:HGroup>
    <s:Label text="Set 3D zTo: "/>
    <s:HSlider id="zToSlider"
      minimum="0" maximum="360" value="0"
      change="textRotationZ.play()"/>
  </s:HGroup>
</s:VGroup>

</s:Application>

```

Summary

Hopefully this was a fun chapter. You learned how to define behaviors in Flex, and we explored several different applications of Flex effects. You learned how to declare effects using MXML, and you tested standalone, sequential, and parallel effect animations. You also learned how to tether behaviors to state changes by defining transitions. Finally, you gained a useful exploratory technique while learning about filter effects by creating a test bed application that allows you to play with different object properties using UI controls.

In the next chapter, we jump into the topic of styling, and you'll learn how you make the appearance of your applications really pop.

STYLING AND SKINNING

“Jessie’s brother stepped off the train trying to remember what a Davis Cup tennis player looked like. He undoubtedly was the first and last passenger ever to step off a Great Northern coach car at Wolf Creek, Montana, wearing white flannels and two sweaters.”

—Norman Maclean,
A River Runs Through It

Most projects will require some degree of styling in order to achieve a desired appearance. Fortunately, Flex applications are very customizable, and the linkage between developers and designers has been softened to make it easier for nonprogramming designers to contribute styles and skins, and for non-designing developers to incorporate the work of designers.

Generally speaking, *styling* involves changing basic style properties—such as font family, text color, and corner radius—that Flex components recognize by default. On the other hand, *skinning* refers to a larger-scale modification of a component’s look and/or behavior. Imagine a **Panel** container skinned to resemble a page in a catalog or a slider control hacked to look like a dial.

In this chapter, you’ll learn how to centralize your style assignments using Cascading Style Sheet (CSS) syntax, an approach that allows you to easily apply a fresh set of styles and rapidly transform the look of your applications. You’ll also learn how to create custom component skins by embedding assets (images or sound effects), applying unique FXG graphics, and incorporating expressive behaviors.

IN THIS CHAPTER

- Inline Style Assignments
- Style Blocks and CSS
- External CSS
- Skinning
- Summary

NOTE

For a great example of skinning and an impressive demonstration of what Flex can do, take a look at Figure 15-1 and check out Audiotool (<http://www.audiotool.com>), an online sound studio that runs on Flash player.

Visit <http://www.creativeapplications.net/> to stay abreast of what others are doing with RIA technologies. The site doesn't cater solely to Flash/Flex productions, and that's a great reason to follow it. After all, you don't want to get tunnel vision.



Figure 15-1. Building the floor of a virtual sound studio at www.audiotool.com, which, besides being a neat RIA, is heavily skinned

Inline Style Assignments

Until now, any styles we've applied were established using inline style assignments. The code in Example 15-1 creates a **BorderContainer** (an ugly one) built with inline styles.

Example 15-1. A **BorderContainer** with inline style assignments

```
<s:BorderContainer id="borderContainer"
    cornerRadius="20"
    borderColor="#00FF00"
    borderWidth="4"
    borderStyle="solid"
    backgroundColor="#0000FF">

</s:BorderContainer>
```

Inline styles are fine if you're working with a relatively small number of components, such as when you're creating proof-of-concept demos and simple widgets.

But what if your application uses several **BorderContainer** components and you want each one to have the same border style and background? Applying these style properties individually would become tedious. More to the point, what if you wanted to change all your style characteristics later? Changing these styles on every individual component becomes ultra-tedious; plus, it becomes difficult to track changes when they're spread throughout your application.

Fortunately Flex supports CSS, which is a superior approach for styling large applications. Because you should be comfortable applying inline property assignments, this chapter is primarily focused on CSS styling and component skinning.

Is That a Style?

There is a semantic tendency to use the words “attribute,” “property,” “style,” “trait,” and even “characteristic” interchangeably. However, in the context of this chapter the distinct meaning of each word is quite important.

Specifically, **id**, **height**, and **width** are component properties, but **fontFamily**, **color**, and **borderStyle** are style attributes. Because CSS definitions apply only to styles, it's important for you to know when a property is a true style and when it's something else, such as a size or an effect property. Here are a few tips for distinguishing between styles and the other property types:

- If you're using Flash Builder in Design mode, switch the Properties pane to Category view and expand the Styles section (Figure 15-2).
- If you're using Flash Builder in Source mode, watch the icons appearing next to the code hints; the icon representing styles looks like blue blocks stacked in an “L” shape (Figure 15-3).

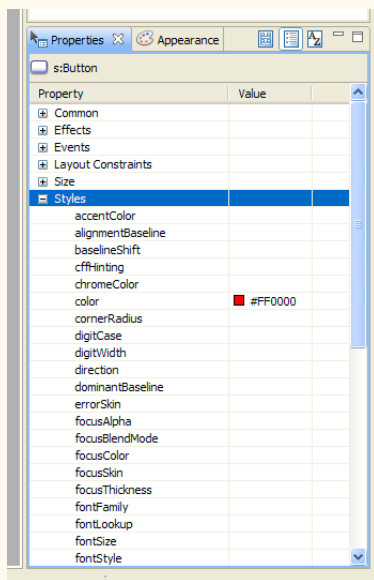


Figure 15-2. Flash Builder's Design mode, with the Properties pane in Category view

- The top authority for component-level documentation is Adobe's Flex 4 Language Reference. Just open the Language Reference and scroll down to the Styles section (Figure 15-4).

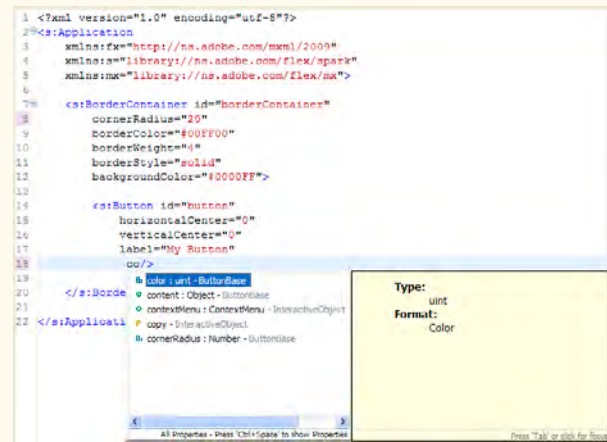


Figure 15-3. Code hinting in Source mode, with icons representing the different property types

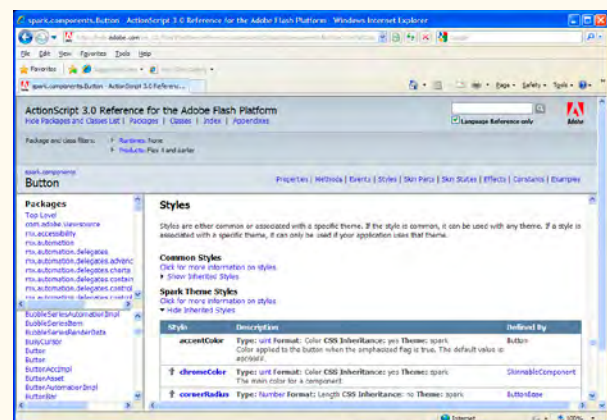


Figure 15-4. Inspecting the styles available to a Spark Button control using Adobe's Flex 4 Language Reference

Style Blocks and CSS

The alternative to inline style assignment is a centralized stylesheet, which can either be nested within a **Style** block or referenced in an external file. First we'll talk about the **Style** block, and then we'll talk about CSS syntax and style definitions.

The Style Block

Regardless of whether you prefer a nested or an external stylesheet, you should declare **Style** blocks somewhere near the top of your application code. For now, pay more attention to the construction of the **Style** block than its nested style rules, which we discuss later. If we converted the previous example into a nested stylesheet, it would resemble Example 15-2.

Example 15-2. A Style block with style assignments for a Spark BorderContainer

```
<fx:Style>
  @namespace s "library://ns.adobe.com/flex/spark";
  @namespace mx "library://ns.adobe.com/flex/mx";

  s|BorderContainer{
    cornerRadius: 20;
    borderColor: #00FF00;
    borderWidth: 4;
    borderStyle: solid;
    backgroundColor: #0000FF;
  }
</fx:Style>

<s:BorderContainer id="borderContainer">

  <!-- This BorderContainer would get its styles from the nested
  style sheet -->

</s:BorderContainer>
```

NOTE

The syntax `s|BorderContainer` is known as a CSS selector. Specifically, it's a type selector, and it applies its style definitions solely to **BorderContainer** objects in your layout. The different CSS selector conditions include the global, descendant, name, and id selectors, which we cover individually later in the chapter.

Conversely, if we referenced the same style definitions in an external CSS file, the Style block would require a **source** assignment and resemble Example 15-3.

Example 15-3. A Style tag with a source attribute defining an external stylesheet

```
<fx:Style source="myStyles.css"/>

<s:BorderContainer id="borderContainer">

  <!-- This BorderContainer would get its styles from an external style
  sheet -->

</s:BorderContainer>
```

NOTE

The only property available to an MXML Style tag is **source**, which defines the location of an external stylesheet.

In this case, the **BorderContainer** would pull its style definitions from an external CSS file, *myStyles.css*. The style definitions in the external file honor the exact same syntax as the nested stylesheet.

CSS Syntax

Your stylesheet should include namespace manifests followed by style definitions, and every style definition should take the form of a selector condition followed by a block of style rules.

Namespaces in CSS

When you open a Style block, Flash Builder will automatically create two namespace definitions for the Spark and Halo component sets; they will be the familiar **s** and **mx** namespace abbreviations, respectively.

You can also use stylesheets to define style rules for custom components, but of course you'll need to make sure to declare a namespace pointing to the component's package location or appropriate URI. If we expanded the previous example to reach custom components under *src/com/learningflex4*, the CSS namespace assignments would resemble Example 15-4.

Example 15-4. CSS namespace assignments, including a namespace for a custom package

```
<fx:Style>
    @namespace s "library://ns.adobe.com/flex/spark";
    @namespace mx "library://ns.adobe.com/flex/mx";
    @namespace lf4 "com.learningflex4.*";

    /* style assignments go here */

</fx:Style>
```

Next, we discuss the various selectors (condition syntax) used by CSS to identify which components will receive which styles.

Style selectors

CSS syntax supports five categories of style definitions, called *selectors*. The main difference between them is how far up the containment ladder a style is declared and thus, how far it's allowed to cascade down a layout.

The five selector categories—global, descendant, name, type, and id—are similar to conditions used to define style targets, and each selector is built using a slightly different syntax. What follows is a description of each selector and an example declaration.

NOTE

By saying “open a Style block,” we mean typing only this much: **<fx:Style>**.

That's the opening tag of the Style block, and when you complete the opening tag by adding the right angle bracket (**>**), Flash Builder will automatically copy all namespaces in use by the application into the stylesheet. The copying will also include any namespaces pointing to custom components and third-party libraries you may be using.

If you provide a **source** property declaring the location of an external stylesheet, Flash Builder will not copy the namespaces.

NOTE

You can declare comments within your stylesheet by wrapping them between the symbols **/*** and ***/**.

Cascading Styles

The style **color**, as well as many other styles, will “cascade” from a parent container down to its children. This arrangement is known as CSS **inheritance**, and it means that you can set a style on a container (or the **Application**), and it will propagate to all the children of that container, setting their text to the same color. By taking advantage of this, you can easily set the text color of most components by setting the **color** style on the **Application**.

Chrome

Some containers have prominent chrome, or graphics that give them a certain “look and feel.” An example of a container with pronounced chrome is the **Panel**. It has a title bar and borders that give it the look of a UI window.

Style Rule Syntax

As you’re reading about CSS selectors, remember that the syntax used to define the style rules will always use the following format:

```
selector{
    style-name: value;
}
```

Note that the style rule opens with the style name, followed by a colon (:), followed by a value, and closed by a semicolon (;).

Also, expect style names to be hyphenated (-) versions of the familiar MXML style properties. Consider the style **chromeColor**. In a CSS style rule, you should reference that property using the style name **chrome-color**:

```
s|button{
    chrome-color: navy;
}
```

You should know the Flex compiler will still process style rules that reference a style’s familiar name (e.g., **chromeColor**), but Flash Builder’s code hinting will suggest only hyphenated style names.

NOTE

These examples define color values as names. While we used this approach lightly in Chapter 9, we haven’t discussed it very deeply. For more info, see the sidebar “Color Names” on page 320.

Global selectors

As its name implies, the global selector defines styles for every component in your application. You probably won’t define many styles globally, although **fontFamily** and some color styles do make appropriate global styles. As demonstrated in the following code, the global selector is distinguished by its use of the reserved name “global”:

```
/* global selector */
global{
    chrome-color: red;
}
```

Descendent selectors

This selector type defines a style based on containment of one component class by another. In the example, notice the selector defines a container, followed by a space, followed by a component. Each component is identified by the combination of namespace abbreviation (**s**), followed by a pipe (|), followed by a component class. Code completion should help you with the construction. The following style would apply to any Spark button contained by a **BorderContainer**:

```
/* descendant selector */
s|BorderContainer s|Button{
    chrome-color: teal;
}
```

Name selectors

The name selector, also called a *class selector*, is so called because you can name it just about anything. Syntax for the name selector begins with a dot (.), followed by a unique style name, followed by a batch of style rules. When you create a name selector, you can define any style/value combinations you want, and later, when you apply the style rule to a component, it will accept rules for any style it supports. You apply name selector styles to various components by setting a component's **styleName** property equal to the name you provided for your name selector (only without the dot):

```
/* name (class) selector */

.specialButtonStyle{
  chrome-color: maroon;
  color: white;
}
```

NOTE

Here's how you would apply a name selector style to a **Button**:

```
<s:Button id="button"
  label="Button"
  horizontalCenter="0"
  verticalCenter="0"
  styleName="specialButtonStyle"/>
```

Type selectors

This is the selector category we used near the start of the chapter. The declaration format for a type selector is namespace abbreviation (**s**, **mx**, etc.) followed by a pipe (|), followed by the component class to be styled. The following example would apply a **chromeColor** (as **chrome-color**) of olive to any Spark Button in the application:

```
/* type selector */

s|Button{
  chrome-color: olive;
}
```

Id selectors

The id selector will apply its style rules to every component in an application having a certain id. Make sure to use this selector wisely, as it could create some confusion if you have multiple components with the same id. The id selector is created by prepending a valid component id with the pound (#) symbol. In the example, the **chrome-color** and the **color** styles will be applied to all components having the **id** "submitButton".

```
/* id selector */

#submitButton{
  chrome-color: navy;
  color: white;
}
```

Color Names

Usually you can specify colors using a hexadecimal value, but in the case of a few colors you can use a special color name instead. For instance, when setting the **backgroundColor** of a **BorderContainer** within its MXML tag, you could specify black in either of two ways:

```
backgroundColor="0x000000"
```

or:

```
backgroundColor="black"
```

This can be convenient for basic colors such as red, green, blue, white, and black, but if you need a special color, it's best to use the Properties pane in Design mode or another color selection utility (they're all over the Web) to determine the correct hex value.

Having said that, the special color names recognized by Flex are "black", "blue", "green", "gray", "silver", "lime", "olive", "white", "yellow", "maroon", "navy", "red", "purple", "teal", "fuchsia", "aqua", "magenta", and "cyan".

In addition to these, Flex also recognizes four custom colors called "haloOrange", "haloBlue", "haloSilver", and "haloGreen".

NOTE

From weakest to strongest association, component styling will observe the following styling precedence. Where the global selector is the weakest, inline styling is the strongest:

- Global
- Descendant
- Name
- Type
- Id
- Inline

Selector precedence

In the previous subheading, we discussed selector categories in the following order: global, descendant, name, type, and id. We discussed selectors in this order because such is their order of precedence, or in other words, how they flow from weakest to strongest style association.

You could test this by creating a simple testing application with a lone **Button** control. Then, if you created a nested stylesheet and included each selector example from the previous section, the id selector would override the other selector categories and actually modify the button. That's because, as stated earlier, id selectors have greater precedence than name selectors, which have greater precedence than type selectors, and so on.

While we're on this topic, inline styling trumps everything. That means if you're managing styles in a stylesheet, you always have the option of adding styles inline. If you do, your inline style assignments will override anything coming from CSS.

External CSS

In this section, we apply what you've learned about stylesheets to the **PhotoGallery** application; only this time, we'll work with an external stylesheet.

First, let's get a cut of the application code. We left this project back in Chapter 12 when we discussed navigation, but since then we've discussed effects. As such, note a few lines of emphasis in Example 15-5 where we added both a drop shadow and a simple fade on the **Image** component.

Example 15-5. The PhotoGallery application mostly as we left it in Chapter 12

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  applicationComplete="photoService.send()">

  <s:layout><s:BasicLayout/></s:layout>

  <fx:Declarations>

    <s:HTTPService id="photoService" url="photos.xml" resultFormat="e4x"
      result="photoList.selectedIndex=0"/>

    <s:XMLListCollection id="photoXMLList"
      source="{photoService.lastResult.photo}"/>

  </fx:Declarations>

  <s:XMLListCollection id="photoXMLList"
    source="{photoService.lastResult.photo}"/>

  <!-- drop shadow added -->
  <s:DropShadowFilter id="dropShadow"/>

</fx:Declarations>

<mx:TabNavigator left="10" top="10" bottom="10" width="200"
  creationPolicy="all">

  <s:NavigatorContent label="Photo List" width="100%" height="100%">
    <s:List id="photoList" height="100%" width="100%"
      dataProvider="{photoXMLList}" labelField="@title"
      change="{thumbList.selectedIndex =
        photoList.selectedIndex}"/>
  </s:NavigatorContent>

  <s:NavigatorContent label="Thumbnails" width="100%" height="100%">
    <s:List id="thumbList" height="100%" width="100%"
      dataProvider="{photoXMLList}"
      itemRenderer="ThumbItemRenderer"
      change="{photoList.selectedIndex =
        thumbList.selectedIndex}"/>
  </s:NavigatorContent>
</mx:TabNavigator>

<!-- Halo simple Fade and DropShadow added -->
<mx:Image id="photoImage" source="{photoList.selectedItem.@image}"
  horizontalAlign="center" verticalAlign="middle"
  left="220" top="10" bottom="10" right="10"
  open="progressBar.visible = true"
  complete="progressBar.visible = false"
  completeEffect="Fade"
  filters="{[dropShadow]}"/>

<mx:ProgressBar id="progressBar" source="{photoImage}"
  width="100" bottom="15" right="30"
  visible="false"/>

</s:Application>
```


Creating and Referencing the CSS File

First things first: let's create a CSS file for the **PhotoGallery** application to pull styles from. To create a brand-new CSS file, right-click on your project's *src* directory and select New→CSS File.

You'll be greeted with a dialog asking you to define both a package and a filename. Leave the package field blank, but specify the name *styles* for the filename and click Finish. Flash Builder should add the .css file extension for you. Place the file in the *default* package under your project *src* folder, and Flash Builder will open the file for you in a new editor window (Figure 15-5).

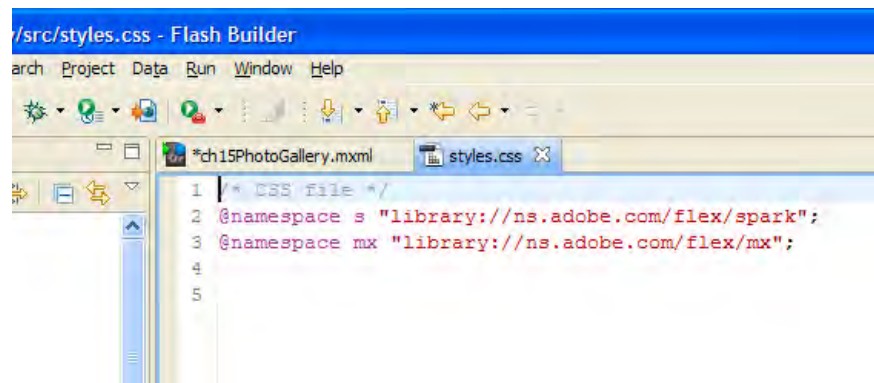


Figure 15-5. Starting from scratch with a fresh external CSS file

The next step is to create a **Style** block and reference the source of our external stylesheet. Jump back to the application code, and just below the **Application** tag, declare a **Style** tag and define its **source** property as shown in Example 15-6.

Example 15-6. Preparing the PhotoGallery application to use an external stylesheet

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  applicationComplete="photoService.send()">

  <fx:Style source="styles.css"/>
```

At this point we'll shift our attention to the CSS file and start adding style rules for the **PhotoGallery** application.

Adding Style Rules to the CSS File

For the sake of demonstration, we'll use each CSS selector category in our stylesheet. Considering the size of the **PhotoGallery** application, proper styling could be accomplished much more aerodynamically with just a couple

rule definitions, but we want to make sure you apply each selector category in a running example, hence this approach.

First, as shown in Example 15-7, let's add some global rules to define **chromeColor**, **selectionColor**, **rollOverColor**, and **focusColor**, all the properties that we want to remain consistent across the application.

Example 15-7. Defining global styles for the PhotoGallery./* CSS file */

```
@namespace s "library://ns.adobe.com/flex/spark";
@namespace mx "library://ns.adobe.com/flex/mx";

global
{
  chrome-color: #389E10;
  selection-color: #389E10;
  roll-over-color: #38C916;
  focus-color: #CAFA6E;
}
```

Next, we'll add a descendant selector to apply a **backgroundAlpha** of **0.0** to any **List** control wrapped in a **NavigatorContent** container that is contained by a **TabNavigator**, as shown in Example 15-8. Of course, this will apply to either of our **List** controls.

Example 15-8. Defining a descendant selector style for the PhotoGallery

```
mx|TabNavigator s|NavigatorContent s|List
{
  content-background-alpha: 0.0;
}
```

Now create the name selector shown in Example 15-9, which we'll assign to the **TabNavigator** control to replace its inline **backgroundColor** property. Note that the style rule is descriptively named **.backgroundStyle**.

Example 15-9. Defining a name selector style for the PhotoGallery

```
.backgroundStyle
{
  background-color: black;
}
```

With the name selector rule defined, jump back into the application code and append the **TabNavigator** control to apply the style named **backgroundStyle**, as shown in Example 15-10.

Example 15-10. Applying the name selector style to the TabNavigator via its **styleName** property

```
<mx:TabNavigator left="10" top="10" bottom="10" width="200"
  creationPolicy="all" styleName="backgroundStyle">
```

We're getting close. Next, we'll apply a type selector to ensure that any instance of a **List** control will use white text, as shown in Example 15-11. If you haven't noticed, we've been creating an increasingly darker **PhotoGallery** application. At this point, if we want to read the unselected labels in either **List** control, we'll want the text to be lighter.

NOTE

We're mentioning these common properties by their familiar names in MXML (e.g., **selectionColor**), but in the stylesheet, remember to convert the lower camel case into hyphenated property names (e.g., **selection-color**).

NOTE

If our **PhotoGallery** had a **List** component contained by a **Panel**, our descendant selector would not target that **List**. Only **List** components wrapped in **NavigatorContent** containers that are wrapped in **TabNavigators** will satisfy the selector's condition.

NOTE

A component's **styleName** property is used to attach a name selector style rule; only don't forget to drop the dot before the style rule's name when you assign the rule to a component. Thus, where the rule is named **.backgroundStyle** in the stylesheet, it should be named simply **backgroundStyle** when it's applied to your component.

Example 15-11. Defining a type selector style for the PhotoGallery

```
s|List
{
    color: #FFFFFF;
}
```

And last but not least, we'll create an id selector to tweak the appearance of the **ProgressBar** component, as shown in Example 15-12. According to style precedence, setting the **chromeColor** inside an id selector will override the value applied earlier in the global selector.

Example 15-12. Defining an id selector style for the PhotoGallery

```
#progressBar
{
    color: #FFFF00;
    chrome-color: #FFFF00;
}
```

If you run the application at this point, you'll find it has a new look, but something's still missing (Figure 15-6).

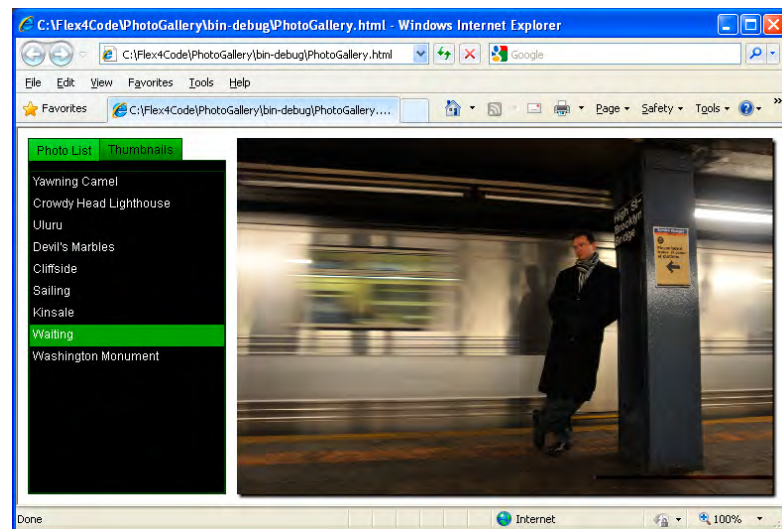


Figure 15-6. The increasingly dark but as yet unfinished PhotoGallery

OK, we'll spill the beans: we're not digging the white application background. We could go with a deep, dark background color, but something about that just seems too plain. What we really want is a gradient fill, something that won't create a boring, flat fill across the page. In the next section, you'll see how to get the fill we want using FXG graphics.

Creating Interesting Backgrounds with FXG Graphics

As our last act with the **PhotoGallery** application, we're going to declare a simple FXG graphic to apply a cool background style.

Since our graphic will be among other elements in the display list, we want to declare it at the bottom of the order. As such, we'll place it between the Declarations section and the MXML components.

Start by opening a new **Rect** graphic and giving it constraints anchored to each corner, like so.

```
<s:Rect left="0" right="0" top="0" bottom="0">
    <!-- more parts go here -->
</s:Rect>
```

Next, inside the **Rect** declaration, add an empty **fill** class; the **fill** won't take any properties:

```
<s:Rect left="0" right="0" top="0" bottom="0">
    <s:fill>
        <!-- more parts go here -->
    </s:fill>
</s:Rect>
```

Now add a **LinearGradient** class inside the fill. By default the **LinearGradient** is rendered from left to right. Since we want a gradient that goes from top to bottom, give the class a **rotation** property of **90**:

```
<s:Rect left="0" right="0" top="0" bottom="0">
    <s:fill>
        <s:LinearGradient rotation="90">
            <!-- more parts go here -->
        </s:LinearGradient>
    </s:fill>
</s:Rect>
```

The **LinearGradient** defines the rendering instructions for our fill, but we still need to define the color elements that will participate in the gradient blend. For this, use the **GradientEntry** class to define each color. As you see in Example 15-13, first create a black, and then create a gray color entry. Each **GradientEntry** will need both an **alpha** and a **color** property assignment.

Example 15-13. The completed FXG fill graphic; the graphic should be added between the Declarations section and the **TabNavigator** control

```
<s:Rect left="0" right="0" top="0" bottom="0">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry alpha="1.0" color="#000000"/>
            <s:GradientEntry alpha="1.0" color="#999999"/>
        </s:LinearGradient>
    </s:fill>
</s:Rect>
```

NOTE

As you may remember from seeing FXG code in the previous chapter, you can also declare a **stroke** class when you want to define a graphic border for FXG objects.

NOTE

If you wanted to simulate a so-called “sun angle” effect, you could use a **rotation** angle of **45** or **135**. An angle of **45** matches the default settings of the **DropShadowFilter**.

NOTE

Other potential fill children available to you include the simple **SolidColor**, the **BitmapFill**, and the **RadialGradient**. The **RadialGradient**, like its linear counterpart, creates a gradient fill with elliptical qualities. The **BitmapFill** generally renders an embedded source image. You can probably guess the effect of the **SolidColor** fill type.

Your **PhotoGallery** application is officially complete and ready to run. Compare your running result to Figure 15-7. In the next section, we discuss skinning and demonstrate how to create custom component skins.

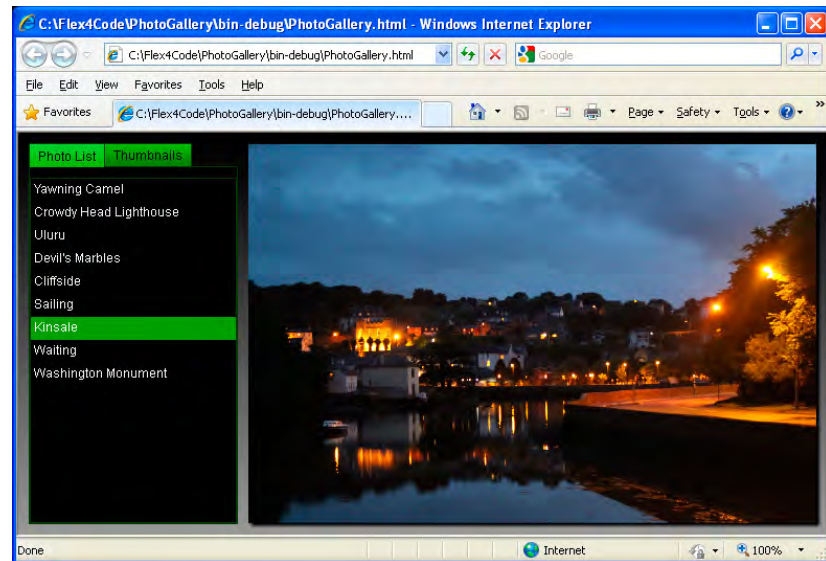


Figure 15-7. The well-dressed PhotoGallery application

Skinning

Our next wave of discussion and experimentation concerns skinning. Just to restate our explanation from the chapter's intro, *skinning* refers to a larger-scale modification of a component's look and/or behavior. Unlike styling, which basically involves applying values *en masse* to various components' style properties, skinning generally implies that new graphic qualities—possibly FXG graphics, images, effects, or filters—are used to remodel some aspect of, or 100% of, a component's look.

We'll jump right into component skinning by having you create a custom skin for the search button in the **YahooSearch** utility. Once you're finished with that task, we'll shift gears and demonstrate how FXG graphics, states, transitions, and effects can really lend something new and fresh to a custom component.

To get rolling, let's make a simple skin for the search button in the **YahooSearch** application, so if you haven't already, go ahead and open that project.

NOTE

We don't really need the **YahooSearch** project's application code to work this example, so we elected not to reprint it this time.

Converting Styles to CSS

If you're so inclined, you can use Design mode to add style rules to a CSS file, which is particularly handy when you want to assign color values. Just follow these steps:

1. To add style rules from Design mode, make sure your Properties pane is in Standard view, select a component for which you want to create a style rule, and then click Convert to CSS (Figure 15-8).

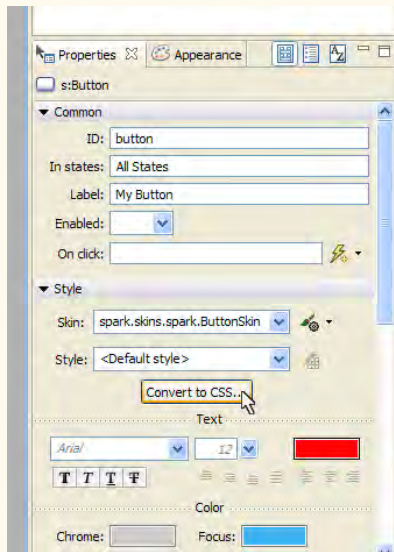


Figure 15-8. The Convert to CSS button in Design mode→Properties pane→Standard view

You should be greeted by the New Style Rule dialog. What happens next depends on whether you already have an external CSS file.

2. If you **do** have a CSS file, just select your CSS file in the drop-down menu labeled "Define style in". But if you **don't** have an external CSS file, you can create one from the New Style Rule dialog by clicking the New button (Figure 15-9).

If you're creating a new CSS file, it's easy. Just provide a name for the file, and, if appropriate, a package location. Packages can be any folder in your *src* directory, but they generally imply reusable source code. Don't, however, add the *.css* file extension—Flash Builder will handle that for you. Once you've created your CSS file, you should have a stylesheet selected in the drop-down menu (Figure 15-10).

3. With a stylesheet selected, you have a few options available for creating style rules. No matter which option you select, the concept is the same: Design mode will take whatever inline styles you have specified and convert them to CSS style rules.

If you just selected a component and applied some styles in the Properties pane, those styles will be converted into CSS style rules. However, existing inline styles will be converted, too, and when you return to Source mode, depending on the sort of style conversion you performed, the inline styles you're used to seeing might be missing.

If at first you become comfortable adding and modifying CSS styles manually, the convenience offered by Design mode's CSS conversion utility will be that much more intuitive and powerful. Therefore, we encourage you to begin by creating style rules manually, in Source mode.

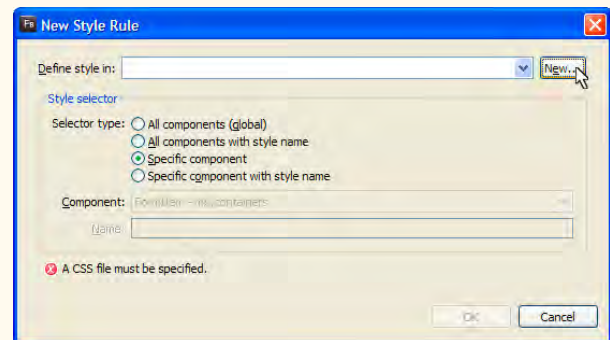


Figure 15-9. The New Style dialog, ready for parameters

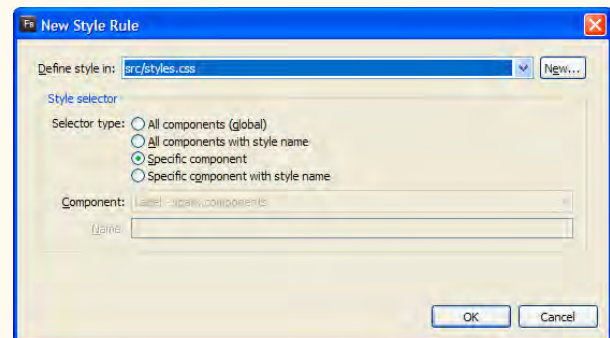


Figure 15-10. Making a new style rule

Working with Packages

We're about to create a custom button skin as a class. As with most custom components, we want to save our skin in an appropriate package. We've already created an external source directory and added some packages to it, so assuming you have the **YahooSearch** application open, and assuming you haven't unlinked the external source path, expand the *[source path]* *com* directory, select *learningflex4*, and then follow New→Package.

When the dialog opens, the "Source path" field should already have a value. Just modify the Name so that it reads *learningflex4.skins*, and click Finish (Figure 15-11).

NOTE

The best practice is to create package names using all lowercase characters.

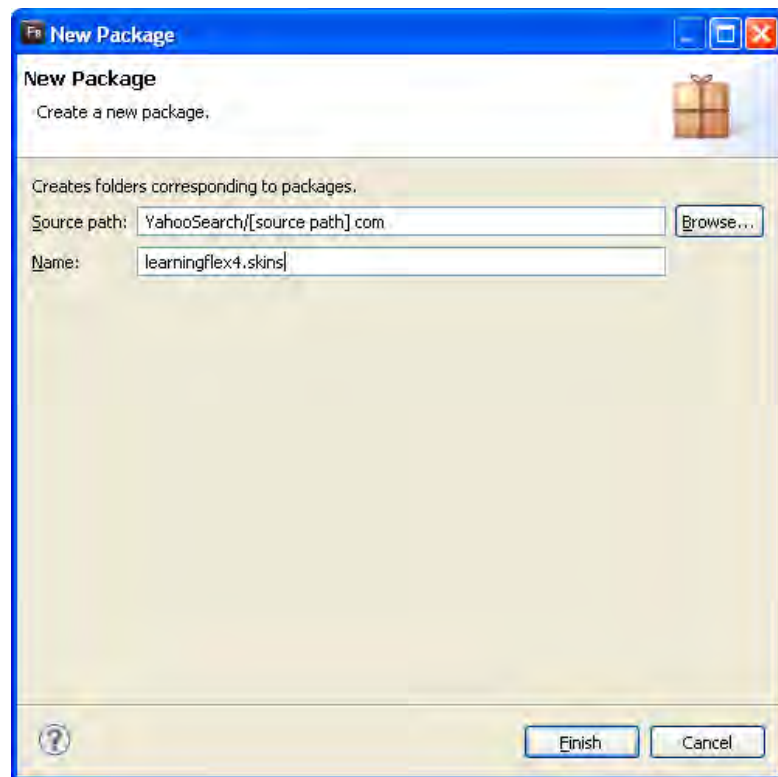


Figure 15-11. Creating a new package for our custom skin classes

In creating our new package, we've continued to observe *reverse domain package naming*. As multiple developers create custom classes, it's just a matter of time before class name conflicts exist. Is it so unlikely that **SearchButtonSkin** doesn't already exist somewhere in the wild? Rather than risk conflicts in the classpath, the solution is to package components in a unique directory that takes the theme *com.domain.class*, where domain is the developer's web domain, and class is the type of class you expect to find in that package. To elaborate, here are a few additional examples of reverse package naming:

com.learningflex4.components

This package would be expected to contain custom component classes.

com.learningflex4.methods

This package would likely contain method classes, which essentially are ActionScript 3 files with public methods that can be reused between projects.

com.learningflex4.skins

Custom skins would likely be found here. You get the idea.

Once your package is created, you'll see it in the Package Explorer (Figure 15-12).

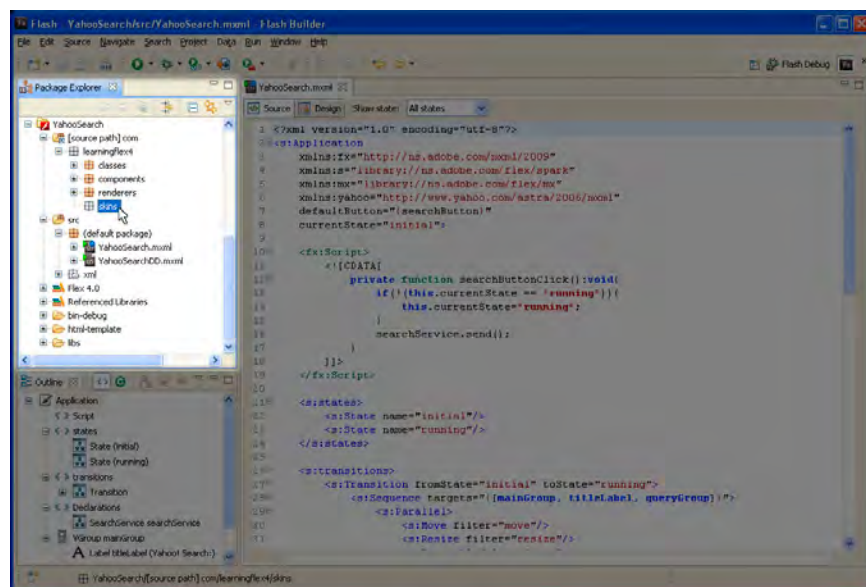


Figure 15-12. Our new skins package appearing in the Package Explorer

Creating a Custom Skin

Now that you have a package to hold your custom skin, right-click/Control-click on the new package and select New→MXML Skin. Like everything else, this will open a new dialog window, New MXML Skin.

Assigning a Host Component

Let's set the *Host Component* first. The Host Component is just an existing Flex component skin class we want to modify to create a custom skin. Click the Browse button to the right of the Host Component field. In the dialog that opens, you may already have a suggested item: "Button – spark.components".

NOTE

The Host Component selection dialog will automatically suggest components it detects in use within your current project.

If the Spark **Button** isn't already suggested, try typing "button" into the input field, and then select the **Button** result that belongs to the Spark components. Make sure *not* to select the Halo **Button**. Alternatively, you could enter a question mark (?) in the search field and get a longer list of components to choose from (Figure 15-13).

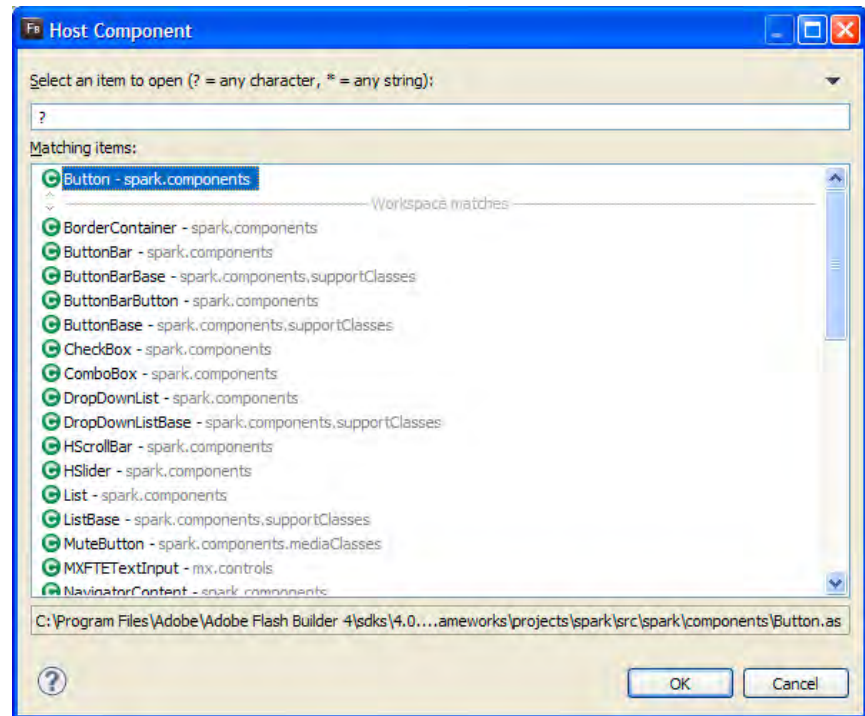


Figure 15-13. Selecting the Spark Button as the Host Component for a custom skin hack

The Package and Host Component fields should now be filled in your New MXML Skin dialog. To finish the configuration, name the skin *SearchButtonSkin*, check the box "Create as copy of", and uncheck the box "Remove ActionScript styling code" (Figure 15-14). When you click Finish, the package should gain a file, and Flash Builder should open that file—which is essentially the shell of the Spark **Button** skin you'll modify to create your custom skin.

At this point, we only need to add a few lines of code to create our custom skin. In the open file editor, scroll down until you find the block of code in Example 15-14; it should be at the bottom of the file.

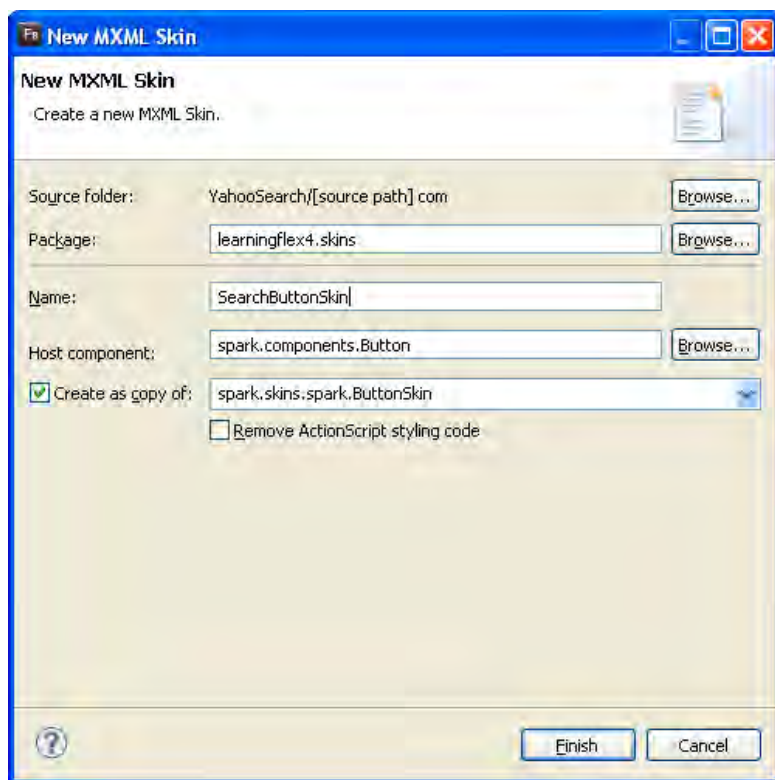


Figure 15-14. Completing the New MXML Skin dialog

Example 15-14. Near the bottom of the code for the Spark Button skin

```
<!-- layer 8: text -->

<!-- OUR STUFF GOES HERE -->

<!-- @copy spark.components.supportClasses.ButtonBase#labelDisplay -->
<s:Label id="labelDisplay"
    textAlign="center"
    verticalAlign="middle"
    maxDisplayedLines="1"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
</s:Label>
```

This last bit of code represents the final portion of the original Spark component, specifically the Spark **Button**'s **Label** text. For our custom skin, we will embed a small magnifying glass icon in the **Button**, just left of the **Label** text, and this is where we'll place the code.

Embedding an asset

To proceed, you'll need to download a PNG graphic from the companion website; you can download it using the following link:

http://www.learningflex4.com/assets/search_icon.png

Once the image is downloaded, either move it or copy it into your *skins* package. This will make it locally available for embedding into your custom skin.

Example 15-15 contains the code necessary to create your first skin.

Example 15-15. *Embedding the magnifying icon into the Button skin as a `BitmapImage` component*

```
<!-- layer 8: text -->
<s:BitmapImage source="@Embed('search_icon.png')"
    height="14" width="14" left="4" verticalCenter="0"/>

<!-- @copy spark.components.supportClasses.ButtonBase#labelDisplay -->
<s:Label id="labelDisplay" paddingLeft="12"
    textAlign="center"
    verticalAlign="middle"
    maxDisplayedLines="1"
    horizontalCenter="0" verticalCenter="1"
    left="10" right="10" top="2" bottom="2">
</s:Label>
```

The most significant portion of our code is this line:

```
<s:BitmapImage source="@Embed('search_icon.png')"/>
```

For the most part, we're just declaring a `BitmapImage` and its `source` location. However, the `source` property also uses the special `@Embed()` directive, which tells the Flex compiler to build the graphic into the compiled SWF. Within the `@Embed()` directive, notice that the asset path is wrapped in single quotes (').

The rest of the skin layout handles size and positioning. Alaric created the original icon as 18 pixels × 18 pixels, but here, by explicitly declaring the icon as 14 pixels × 14 pixels, we're proportionately resizing it to fit our skin. The remaining settings—`left` and `verticalCenter` in regard to the `BitmapImage`, and `paddingLeft` in regard to the `Label`—should make sense.

Applying a custom skin in Source mode

We now have a custom skin, `SearchButtonSkin`, but we still need to attach it to a component. As usual, we'll take advantage of Flash Builder's code completion to help us with this task. Back in the `YahooSearch` application code, find the declaration for `searchButton`:

```
<s:Button id="searchButton" label="Search"
    click="searchButtonClick()"/>
```

After the `click` attribute, add a carriage return and start typing:

```
skinClass="searchbut
```

Now use Ctrl-space bar to toggle code hinting. You should see your newly created skin class available for selection, courtesy of Flash Builder (see Figure 15-15). Go ahead and select it. That should complete the search button's declaration, but look at Example 15-16 to see what you ended up with for the `skinClass` property assignment.

Example 15-16. Assigning the custom skin to a Button using the `skinClass` property

```
<s:Button id="searchButton" label="Search"
    click="searchButtonClick()"
    skinClass="learningflex4.skins.SearchButtonSkin"/>
```

That's a fully qualified path. Also, if you scroll to the top of your Script/CDATA block, notice the new arrival:

```
<fx:Script>
    <![CDATA[
        import learningflex4.skins.SearchButtonSkin;
```

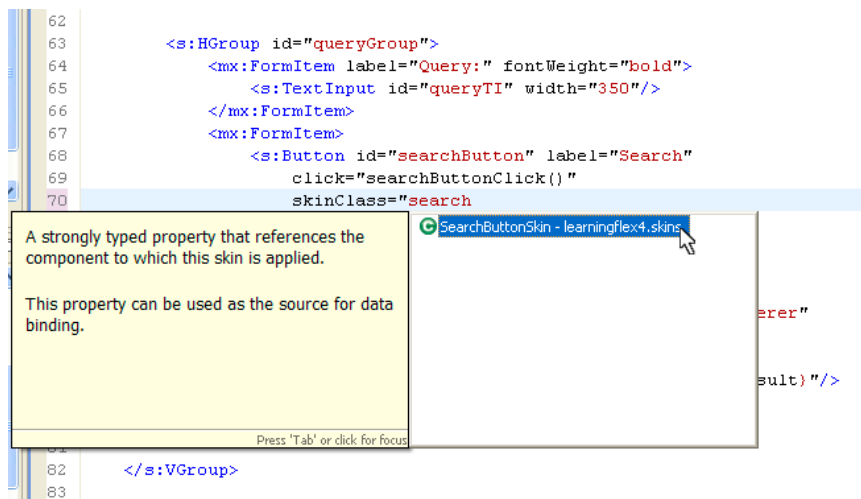


Figure 15-15. Applying the custom skin class using code completion

Besides creating the fully qualified path to your custom skin, code completion also imported the skin class into your project. Assuming you're ready to see your handiwork, recompile the application and give it a spin (Figure 15-16).

Yahoo! Search:

Query:

Figure 15-16. The skinned search button, in all its graphical glory

Applying a custom skin in CSS

In the previous section we used an inline style assignment to attach the custom skin to our search button, but you can also handle this with CSS. Assuming you have either a nested or an external stylesheet ready and waiting, the code in Example 15-17 uses a CSS **id** selector to attach the custom skin to the search button.

Example 15-17. Using a CSS id selector to attach *SearchButtonSkin* to the *searchButton* component

```
/* CSS file */
@namespace s "library://ns.adobe.com/flex/spark";
@namespace mx "library://ns.adobe.com/flex/mx";

#searchButton
{
    skinClass: ClassReference("learningflex4.skins.SearchButtonSkin");
}
```

NOTE

*This is a decent example of when to use the **id** selector in a stylesheet. Unless your application has more than one instance of a search button, there's no reason why you'd want to use a type selector to apply this skin to every button in your application.*

Applying a custom skin in Design mode

If you prefer working in Design mode, you can also apply skin classes through the Properties pane in Standard view. First, make sure the **Button** is selected, and then expand the Skin menu in the Style area. Flash Builder should automatically detect the presence of your custom skin, making it available for selection (Figure 15-17).

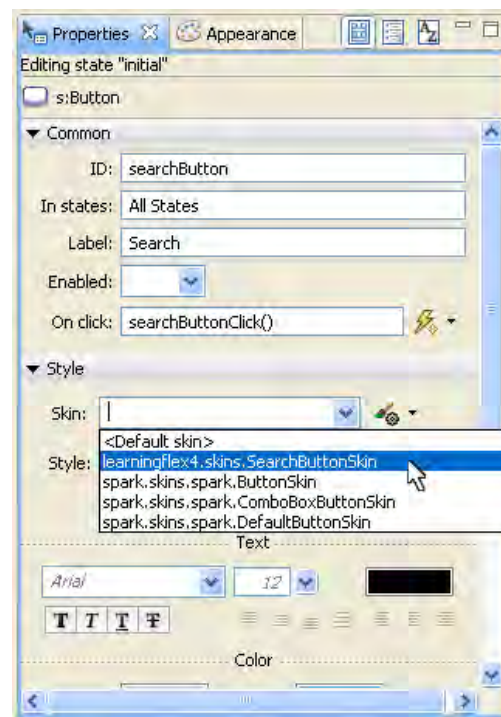


Figure 15-17. Selecting the button skin in Design mode→Properties pane→Standard view

Skinning with FXG Graphics and Effects

Now that you've seen how to create a simple skin quickly, let's throw a few extra tricks into the mix by creating a **Button** that renders an animated FXG graphic.

For this exercise, start a new project called **BusySkin**. Once the new project is created, add the external source path by following Project→Properties→Flex Build Path→Source path. In this dialog, select Add Folder and browse to your external source; we've been keeping ours in `C:\Flex4Code\com`. Now you should be ready to get rolling.

Expanding the package directory

This time we're going to skin with an FXG graphic. We don't want to start a habit of adding assets to the same directory as our custom skins every time we make a skin, so let's expand the package to accommodate a graphics class. In your Package Explorer pane, right-click/Control-click the skins package and select New→Folder. When the dialog appears, type in the name *graphics*.

Adding the FXG graphic

With your expanded package directory, you're ready to acquire the FXG graphic for this exercise, which you can download from the companion website: <http://www.learningflex4.com/fxg/CoolGraphic.fxg>.

You can either copy and paste the code into a new FXG file or save the file directly into your package structure; just make sure to save it in the package's graphics folder.

We'll provide the FXG code here in Example 15-18 so it's available for those who might not have convenient access to the Internet.

Example 15-18. *CoolGraphic.fxg*

```
<Graphic version="2.0"
  xmlns="http://ns.adobe.com/fxg/2008"
  xmlns:fxg="http://ns.adobe.com/fxg/2008"
  xmlns:d="http://ns.adobe.com/fxg/2008/dt"
  viewHeight="250" viewWidth="250">

  <Ellipse width="228" height="225" x="4.05" y="3.35"
    scaleX="1.0642675161361694" scaleY="1.078149676322937">
    <fill>
      <LinearGradient x="263.1" y="20.45"
        scaleX="347.5792463597331" rotation="148.10469836029338">
        <GradientEntry color="#75AE51" ratio="0.006756756756757"/>
        <GradientEntry color="#B5B1B0" ratio="0.9864864864864865"/>
      </LinearGradient>
    </fill>
```

NOTE

To create an empty FXG file, right-click/Control-click the *graphics* folder and select New→File. When you name the file, make sure to include the *.fxg* file extension.

```

<stroke>
  <LinearGradientStroke x="260.25" y="23.2" weight="2.8"
    scaleX="342.4026613506385" rotation="146.82741369772396">
    <GradientEntry color="#FEFDFD" ratio="0.013513513513513514"/>
    <GradientEntry color="#A2A1A1" ratio="0.9932432432432432"/>
  </LinearGradientStroke>
</stroke>
<filters>
  <GlowFilter blurX="8" blurY="8" color="FFFFFF" alpha="1"
    strength="2.4" quality="2" inner="false" knockout="false"/>
</filters>
</Ellipse>

<Ellipse width="59" height="60" x="47" y="42">
  <fill>
    <LinearGradient x="-9" y="-6" scaleX="105.38026380684383"
      rotation="43.846549548894636">
      <GradientEntry color="#EDA74A" ratio="0.02027027027027027"/>
      <GradientEntry color="#FBFAF1" ratio="0.9864864864864865"/>
    </LinearGradient>
  </fill>
  <stroke>
    <SolidColorStroke color="#FEF9F8" alpha="1" weight="0.7"/>
  </stroke>
</Ellipse>

</Graphic>

```

That should complete the graphic import. Now it's time to use it in a skin.

About skin class requirements

You've seen how to associate a custom skin with a Host Component, but skin classes have some other requirements, namely required states and required parts.

Different components have different *required skin states*, and even if you don't apply transitions or state-specific properties, you still have to declare each required state for a certain skin type.

Like required states, component skins also have *required skin parts*, which may include graphical elements and/or other objects essential to the skin's Host Component.

Since each component might have a different array of skin requirements, how can you uncover the nitty-gritty details each time you prepare to make a custom skin class? Well, to start, like so much other information, each Flex component's skin requirements are described in the Language Reference (Figure 15-18). But there's also another helpful hinting system you're about to discover.

Creating the GraphicButtonSkin

Right-click/Control-click on the skins directory in your custom package and select New→MXML Skin. Like last time, set the Host Component to **spark.components.Button**. Name the skin *GraphicButtonSkin*. This time, however, uncheck the option "Create as copy of", and then select Finish (Figure 15-19).

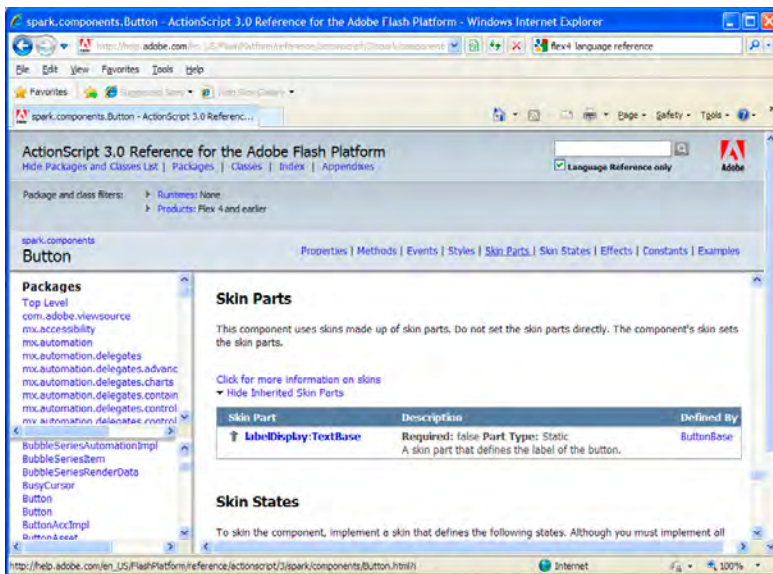


Figure 15-18. Researching skin requirements in the Language Reference

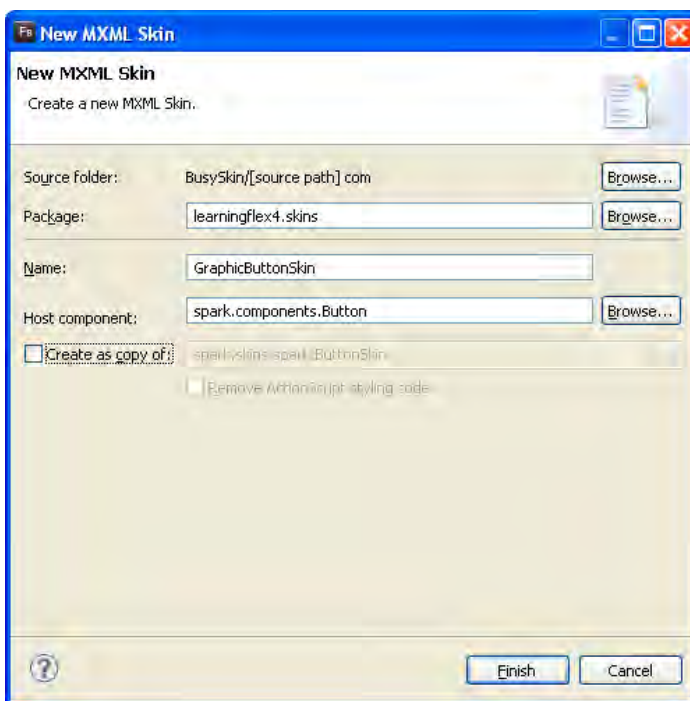


Figure 15-19. The New MXML Skin dialog configured to create a skin shell, which has no code overhead

This is a little different. Compared to last time, there's definitely a lot less code lingering around. Let's look at Example 15-19 to see the code in print.

Example 15-19. *Creating a custom Button skin starting from “shell” code*

```

<?xml version="1.0" encoding="utf-8"?>
<s:Skin
  xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx">

  <!-- host component -->
  <fx:Metadata>
    [HostComponent("spark.components.Button")]
  </fx:Metadata>

  <!-- states -->
  <s:states>
    <s:State name="disabled" />
    <s:State name="down" />
    <s:State name="over" />
    <s:State name="up" />
  </s:states>

  <!-- SkinParts
  name=labelDisplay,
  type=spark.components.supportClasses.TextBase,
  required=false
  -->
</s:Skin>

```

Just looking at the shell code, we see the **Button** skin class has four required states and one skin part. That’s fantastic; everything we need is right here.

The concept behind GraphicButtonSkin

Basically, we want to skin a **Button** that animates a graphic on **mouseover**. That means we’ll take advantage of the **mouseover** event to trigger a state change transition that will toggle the graphic animation. We’re also going to use the state change to engage a **GlowFilter**. Fortunately, from a development perspective, this means we can arrange our code relative to one required state—**over**, that is, **mouseover**. At any given time, the skin will reflect either the **over** state or, when disengaged, the default state.

Creating the base state

Just below the **SkinParts** comment block, which we included for reference, add the code in Example 15-20 to create the guts for our custom skin. There shouldn’t be any real surprises. Note that both instances of our background graphic element are nearly identical; only their state assignments distinguish them from one another. The first graphic background will serve as the slightly plain base state, and it will be *excluded* from the **over** state. Meanwhile, the second graphic background will apply a **GlowFilter** effect, and it will be *included* in the **over** state. Of course, we haven’t yet declared the **GlowFilter** class.

Example 15-20. Adding background graphics and components to the custom skin

```
<!-- SkinParts
name=labelDisplay,
type=spark.components.supportClasses.TextBase,
required=false
-->

<s:layout>
  <s:BasicLayout/>
</s:layout>

<!-- base appearance -->
<s:Rect top="1" bottom="1" left="1" right="1" radiusX="2"
  excludeFrom="over">
  <s:fill>
    <s:LinearGradient rotation="90">
      <s:GradientEntry
        color="0xC2C2C2"
        alpha="0.85" />
      <s:GradientEntry
        color="0x000000"
        alpha="0.85" />
    </s:LinearGradient>
  </s:fill>
  <s:stroke>
    <s:SolidColorStroke color="#000000" weight="1"/>
  </s:stroke>
</s:Rect>

<!-- identical, but with a glow filter -->
<s:Rect top="1" bottom="1" left="1" right="1" radiusX="2"
  includeIn="over">
  <s:fill>
    <s:LinearGradient rotation="90">
      <s:GradientEntry
        color="0xC2C2C2"
        alpha="0.85" />
      <s:GradientEntry
        color="0x000000"
        alpha="0.85" />
    </s:LinearGradient>
  </s:fill>
  <s:stroke>
    <s:SolidColorStroke color="#000000" weight="1"/>
  </s:stroke>
</s:Rect>

<graphics:CoolGraphic id="graphic"
  scaleX=".045"
  scaleY=".045"
  left="6"
  verticalCenter="0"/>

<s:Label id="labelDisplay"
  text="Button"
  color="#FFFFFF"
  left="28"
  verticalCenter="1"/>
```

NOTE

It took some “playing” with the scale values (`scaleX` and `scaleY`) to get the graphic size we wanted. One great thing about programming is that you never fully lose some reliance on trial and error, and sometimes that little bit of the unknown can really drive your curiosity and help push you forward.

Adding transitional animation and effects

With the skin's layout defined, let's add our **transitions** and **Declaration** blocks; these should be placed above the **SkinParts** comment block, included yet again in Example 15-21 for placement reference.

Our transition—which will be triggered by any state change to the **over** state—will call only one effect, **Rotate3D**, with its hardcoded target, **graphic**, which refers to the **id** of our FXG graphic. Our **Declarations** block includes one lone **GlowFilter** definition. No surprises here; we've seen a lot of code like this.

Example 15-21. Adding a state change transition and a *GlowFilter* declaration to the custom skin

```
<s:transitions>
  <s:Transition fromState="*" toState="over">

    <s:Rotate3D id="rotate" target="{graphic}"
      angleZFrom="0" angleZTo="360" duration="2000"
      autoCenterTransform="true"/>

  </s:Transition>
</s:transitions>

<fx:Declarations>

  <s:GlowFilter id="glow" color="#00FF00" strength="4"/>

</fx:Declarations>

<!-- SkinParts
name=labelDisplay,
type=spark.components.supportClasses.TextBase,
required=false
-->
```

Finally, let's make sure the second background graphic is set up to use the **GlowFilter** effect; this will help to distinguish the **mouseover** condition:

```
<!-- identical, but with a glow filter -->
<s:Rect top="1" bottom="1" left="1" right="1" radiusX="2"
  includeIn="over" filters="{[glow]}">
  <s:fill>
```

Complete code for the custom skin class

That should complete the exercise. For the sake of showing you everything in continuity, Example 15-22 provides the entire skin class.

Example 15-22. The entire skin class for *GraphicButtonSkin*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/mx"
  width="70" height="20"
  xmlns:graphics="com.learningflex4.graphics.*">
```

```

<!-- host component -->
<fx:Metadata>
    [HostComponent("spark.components.Button")]
</fx:Metadata>

<!-- states -->
<s:states>
    <s:State name="disabled" />
    <s:State name="down" />
    <s:State name="over" />
    <s:State name="up" />
</s:states>

<s:transitions>
    <s:Transition fromState="*" toState="over">
        <s:Rotate3D id="rotate" target="{graphic}"
            angleZFrom="0" angleZTo="360" duration="2000"
            autoCenterTransform="true"/>
    </s:Transition>
</s:transitions>

<fx:Declarations>
    <s:GlowFilter id="glow" color="#00FF00" strength="4"/>
</fx:Declarations>

<!-- SkinParts
name=labelDisplay,
type=spark.components.supportClasses.TextBase,
required=false
-->

<s:layout>
    <s:BasicLayout/>
</s:layout>

<!-- base appearance -->
<s:Rect top="1" bottom="1" left="1" right="1" radiusX="2"
    excludeFrom="over">
    <s:fill>
        <s:LinearGradient rotation="90">
            <s:GradientEntry
                color="0xC2C2C2"
                alpha="0.85" />
            <s:GradientEntry
                color="0x000000"
                alpha="0.85" />
        </s:LinearGradient>
    </s:fill>
    <s:stroke>
        <s:SolidColorStroke color="#000000" weight="1"/>
    </s:stroke>
</s:Rect>

<!-- identical, but with a glow filter -->
<s:Rect top="1" bottom="1" left="1" right="1" radiusX="2"
    includeIn="over" filters="{[glow]}">

```

```

        <s:fill>
            <s:LinearGradient rotation="90">
                <s:GradientEntry
                    color="0xC2C2C2"
                    alpha="0.85" />
                <s:GradientEntry
                    color="0x000000"
                    alpha="0.85" />
            </s:LinearGradient>
        </s:fill>
        <s:stroke>
            <s:SolidColorStroke color="#000000" weight="1"/>
        </s:stroke>
    </s:Rect>

    <graphics:CoolGraphic id="graphic"
        scaleX=".045"
        scaleY=".045"
        left="6"
        verticalCenter="0"/>

    <s:Label id="labelDisplay"
        text="Button"
        color="#FFFFFF"
        left="28"
        verticalCenter="1"/>

</s:Skin>

```

NOTE

It's important that the `Label` skin part have an `id` of `labelDisplay`. Without it, you won't be able to define labels for your skinned buttons.

To see your new custom skin in motion, simply attach it to a **Button** component using one of the three tactics described in the previous section. The simple declaration in Example 15-23 applies the skin to a **Button** component using inline syntax and the **Button**'s `skinClass` property. Figure 15-20 shows the result.

Example 15-23. A basic application with a **Button**, just enough to see the custom skin at work

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            import learningflex4.skins.GraphicButtonSkin;
        ]]>
    </fx:Script>

    <s:Button id="button"
        label="Button"
        horizontalCenter="0"
        verticalCenter="0"
        skinClass="learningflex4.skins.GraphicButtonSkin"/>

</s:Application>

```

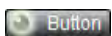


Figure 15-20. The finished **Button** skin (but you'll have to program it and run it to see the animation firsthand)

Thoughts About Themes

Another approach to managing appearances in Flex applications involves finding and applying or creating **themes**. Themes are collections of thematically related style properties and component skins, arranged together to create a unique and aesthetic interface. They can range from simple and elegant to exotic and flashy.

You can find Flex themes posted here and there on the Web, but be careful with them. Themes may be available for purchase, or they may be free, provided you agree to some particular licensing regulations and fine print.

If you want to get a quick impression of themes, a few demos come installed with Flash Builder. To test some themes against your layouts, switch to Design mode→Appearance pane, and where it says “Project theme: Spark”, click on Spark to open the themes portal.

One caveat concerning themes: if you’re applying several low-level style assignments, those might override the theme properties and prevent the full impact of the theme.

Summary

This has been a monster chapter. We congratulate you for hanging in there. By now, you should be comfortable applying styles to your applications using either inline style assignments, Cascading Style Sheets, and/or unique component skins.

You’ve used stylesheets to define style properties at the application level, according to containment order, by component class type, arbitrarily using a style name, and even by component **id**. You know that the benefits of CSS styling include centralizing your styles in one convenient place as well as the ability to quickly swap in a whole new set of style rules in one motion.

You’ve also learned how to skin components by performing simple modifications on copies of Spark component skins as well as by creating your own unique skins. You know how to embed icon graphics, leverage FXG graphics, and even add effects using state change transitions to create rich, engaging component skins.

In the next chapter, we create a simple **ContactManager** application for the purpose of answering the question “How can I get from an input field on a Flex application to a row in a database, and back?”

MAKING DATA DYNAMIC: LINKING FLEX TO THE SERVER

In this chapter, we build an Adobe AIR application called **ContactManager** that integrates Flex with server technologies to handle three basic data management tasks, specifically retrieving, inserting, and editing data. We introduce you to a few “extras” along the way—such as “getters” and “setters” and the inline conditional—but this chapter strives to answer the following two questions:

“How can I load database information into a Flex application?”

“How can I send inputs collected in a Flex application to a database?”

IN THIS CHAPTER

Some Background
Information

The ContactManager
Application

Linking ContactManager
to the Server Using the
HTTPService Class

Summary

Some Background Information

Before now, each chapter in this book has been wholly dedicated to some aspect of Flex development. In this chapter, though, we’re asking you to split your attention between some potentially unfamiliar concepts, including server technologies and Adobe AIR development. To properly set the stage, we want to take a moment to address a few expectations we have regarding your understanding of server technologies as well as some background information concerning Adobe AIR.

Our Choice of Server Technologies: PHP and MySQL

When it comes to server technologies, there’s no shortage of options available to you, and that fact made it difficult for us to decide which platforms to discuss. We believe a significant portion of our readership will be hobbyists and do-it-yourself developers; therefore, we’re presenting a custom Flex integration using PHP and MySQL. Because these platforms are widely available among low-cost web hosting packages, and because they’re free to acquire for anyone interested in creating a home or office development environment,

NOTE

Appendix A, which demonstrates how to arrange a local development environment, also describes how to add the PHP Development Tools (PDT) Eclipse plug-in to your Flash Builder IDE. If you add the PDT plug-in, you'll not only benefit from color-coded PHP syntax, but you'll also get Eclipse XML Tools, which provides both a graphical Design view for working with XML markup and color-coded XML syntax.

NOTE

If you're familiar with JavaScript or Ajax, you might be interested to know that you can create Adobe AIR applications entirely with HTML, JavaScript, and Ajax. For more information, check out the Adobe AIR for JavaScript Developers Pocket Guide, by Mike Chambers et al. (<http://oreilly.com/catalog/9780596518370/>). What's better? O'Reilly and Adobe have made this book a free download: <http://onair.adobe.com/files/AIRforJSDevPocketGuide.pdf>.

NOTE

If you want to use AIR to access the local filesystem or the clipboard, we encourage you to get a book dedicated to AIR development. We recommend Adobe AIR in Action, by Joseph Lott et al. (Manning); Adobe AIR Programming Unleashed, by Stacy Tyler Young et al. (Sams); or Adobe AIR 1.5 Cookbook, by David Tucker et al. (<http://oreilly.com/catalog/9780596522513/>).

Do note that these books all discuss AIR 1.5 or earlier, and at the time of this writing, Adobe has released AIR 2.0. So you'll want to supplement any title you acquire with Adobe's official documentation.

there is a clear financial motive in our selection. Moreover, the PHP/MySQL pairing also benefits from a tremendous volume of documentation in print as well as online, and it's difficult to overestimate the value this brings to anyone learning these technologies.

Because this chapter covers the Flex aspects of a custom integration without explaining PHP or MySQL specifics, you'll need a rudimentary understanding of PHP and MySQL in order to appreciate the chapter's exercise. However, if you're *not* familiar with PHP or MySQL, we've included three appendixes to help you get started. So, if you need a primer on arranging a local development environment, working with MySQL, or writing and calling PHP code, you might want to refer to Appendixes A, B, and C before continuing with this chapter.

Adobe AIR

Working with Adobe AIR shouldn't present any surprises; after all, it's the same Flex we've been learning throughout this book. However, AIR provides functionality and features that aren't available to web-based Flex applications, so we want you to see how easy it is to get started with AIR, even if we don't expose you to the full range of possibilities. For instance, with AIR you can access the local filesystem to create, edit, and save files; also, AIR allows you to use the clipboard to copy and paste content from an external format—such as MS Excel, Safari, or OpenOffice—into a Flex application. AIR even provides a local SQL database, called SQLite, for applications that require basic SQL support.

In this chapter's exercise, we'll start a new Adobe AIR project, which uses a **WindowedApplication** container, and give the system window a **title**—just three lines of code. In Chapter 17, we go a step further and teach you how to assign a custom window icon and create an Adobe AIR installer, but that's about as deep as we'll venture into AIR development. Our goal isn't to give you a deep tour of AIR as much as it is to introduce you to AIR. Most books focused on AIR development expect you to know something about Flex when you get started, so we would like to think we're giving you a head start.

Why Use PHP and MySQL if AIR Provides SQLite?

In our discussion of Adobe AIR, we mentioned AIR has the SQLite database built into the framework. So why would we show you PHP and MySQL if AIR can handle both functions internally? The simple answer is this: we want this lesson to demonstrate skills that will transfer easily to web-based applications, which cannot benefit from SQLite but can use the same **HTTPService** linkage to connect to the services layer.

The ContactManager Application

We present this chapter's exercise in two phases. In the first phase, you'll create the layout and basic functions for the **ContactManager** application as it relates to concepts we've already discussed in previous chapters. In the second phase, you'll extend the application to call server-side PHP scripts that will handle data requests and data submissions to a MySQL database.

Creating an Adobe AIR Project

The first step toward creating an Adobe AIR application requires creating a new Flex project to run in Adobe AIR. You can do this in Flash Builder by customarily following File→New→Flex Project. When the New Flex Project dialog appears, name the project **ContactManager**; then, under Application type, select Desktop (runs in Adobe AIR). Also, since we'll use PHP as our server solution, find Server technology (toward the bottom) and select PHP as the Application server type. With these settings in place, click Next (Figure 16-1).

WARNING

This example serves to demonstrate the minimum code necessary to move data between a Flex application and a MySQL database using PHP. The PHP scripts we present do not provide tight security or error checking; as such, this example is not a production solution. In contrast, a safe, dependable PHP implementation is one that protects against SQL injection vulnerabilities and provides error handling to safeguard the script from malicious input or faults encountered when calling the database. Clearly, these concepts are broad and warrant a focused study of PHP and MySQL.

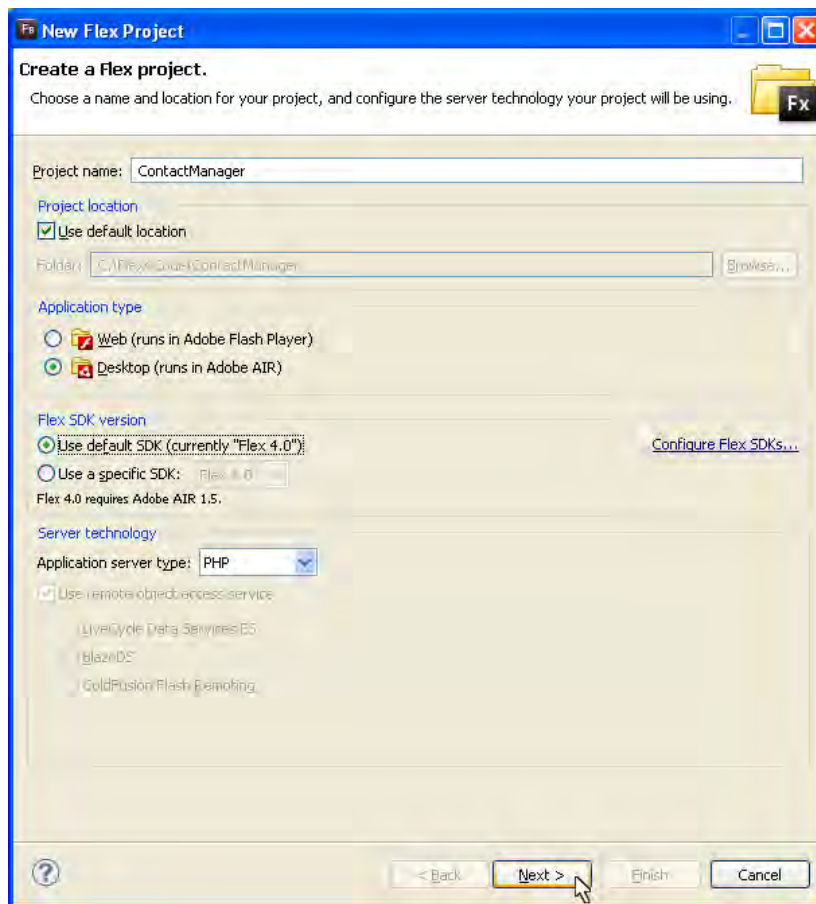


Figure 16-1. Creating a Flex project to run in Adobe AIR

Configuring the Server and bin Locations

Since we chose PHP as the Server technology, Flash Builder wants us to specify server details as well as an external directory for the *bin-debug* path. In order to complete this step, you need to have a valid development environment. If you do not already have a development environment running on your system, follow the steps outlined in Appendix A, which explains how to configure your system to use WAMP (Windows) or MAMP (Mac OS). Figure 16-2 shows how we configured the server location details for a default WAMP installation.

WARNING

WAMP/MAMP needs to be running (i.e., online) for the Validate Configuration test to work. On another note, we've noticed that Skype has a tendency to conflict with WAMP. So if everything seems correct and yet validation continues to fail, try exiting any software, such as Skype, that might be competing with WAMP for control of the Internet ports, and then restart WAMP.

NOTE

You don't have to install WAMP/MAMP to create a local development environment. Rather, you have the option of installing Apache, MySQL, and PHP individually. If you're more interested in developing the forward-facing application than managing the server backend, though, WAMP/MAMP makes it easy to get all three platforms in one pass using a single installer. This is definitely the recommended approach if you're unfamiliar with server technologies.



Figure 16-2. Adding the Server location and Output folder settings

In Figure 16-2, note that we shortened the default name of the Output folder to *ContactManager* by deleting the *-debug* suffix. This will become important later when we define paths to our PHP files. We also set the Output folder (i.e., *bin-debug* in the Package Explorer) to reside just below the Web root directory. In other words, we're asking Flash Builder to compile our application within the web scope of our server environment.

The *web scope* refers to any directories among the local filesystem recognized by the server technology for exposure to server-side processing. In this case, any files added below `C:\wamp\www` would be within the web scope; however, if we created the directory `C:\wamp\local`, it would be outside of the web scope, and files in that directory would be ignored by the server. *It is crucial that PHP files be placed within the web scope in order to be compiled and processed by the server.* Having said that, setting the Output folder to a path within the web scope allows Flash Builder to copy any PHP files you create directly to an external bin, making it easy for you to run and debug your application.

With these settings in place, select Finish to complete the project configuration. Now, Flash Builder will not only compile to but automatically run/debug your application from a directory where your PHP scripts will receive server processing. This is quite handy, as it saves you from a lot of file copying back and forth between the bin and a directory within the web scope every time you recompile and want to test your application.

Code for the ContactManager

Once you create the Adobe AIR project, a couple of differences should stand out. The first one is the use of the **WindowedApplication** container, which provides access to additional features available to AIR applications. The second difference is the presence of a *ContactManager-app.xml* configuration file, which provides access to a few specialty settings, such as the initial size of the application window, whether or not it can be resized, where it's positioned onscreen when it loads, and icons used in the operating system's various menus. We look at the AIR configuration file more closely in Chapter 17.

Examples 16-1 and 16-2 provide the code for the first phase of the **ContactManager** application. If you briefly glance through the source, you'll find it unites several examples from prior chapters. The **ContactManager** application container (Example 16-1) loads the *contacts.xml* file from the companion website into a **DataGrid** control, as we presented in Chapter 11. The **ContactDetailComponent** (Example 16-2) combines elements from Chapter 13 (view states), Chapter 10 (rich forms), and Chapter 14 (effects and transitions) to create a component that doubles as both a details viewer and a details editor; there's also a fair amount of ActionScript (Chapters 5, 7, and 8) handling the decisions and logic. So we're merging several lessons for this exercise.

Go ahead and add the code to get underway. Because you'll be declaring the detail component (Example 16-2) in the main application (Example 16-1), it may help to create the detail component first. Remember, to create a composite component, select the default package of the **ContactManager** project, and then open the File menu and follow New→MXML Component. Base the **ContactDetailComponent** on the **Group** container.

WARNING

PHP files must be added to and called from the web scope in order to receive server-side processing. The web scope refers to any directories within the host filesystem recognized by the server configuration for server-side processing.

NOTE

If you don't want to enter all the code necessary to start this exercise, just download the chapter's source code from the companion website (<http://www.learningflex4.com/code/ch16/ContactManager.fxp>).

NOTE

The `title` property of the **Windowed Application** container sets the title for an AIR application window. If you don't set this property, it will default to the name of the main application file.

NOTE

The `contactsDGItemSelect()` function assigns values to several **Bindable** variables defined in the **ContactDetailComponent**.

NOTE

If you don't have an Internet connection, or if you prefer to use a local file for development, you can use the `contacts.xml` file we made in Chapter 11 and reference it here relative to the project workspace. However, we're about to replace the XML file with a PHP solution, so you won't need it for very long.

Example 16-1. *ContactManager\src\ContactManager.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:WindowedApplication
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    xmlns:local="*"
    creationComplete="refreshApp()"
    minHeight="400" minWidth="700"
    title="Contact Manager">

    <fx:Script>
        <![CDATA[

            public function refreshApp():void{
                contactsDG.selectedIndex = 0;
                contactsDGItemSelect();
            }

            private function contactsDGItemSelect():void{
                detailComponent.contactID =
                    (contactsDG.selectedIndex + 1).toString();
                detailComponent.type = contactsDG.selectedItem.@type;
                detailComponent.firstName = contactsDG.selectedItem.firstName;
                detailComponent.lastName = contactsDG.selectedItem.lastName;
                detailComponent.email = contactsDG.selectedItem.email;
                detailComponent.phoneNum = contactsDG.selectedItem.phone;
            }

        ]]>
    </fx:Script>

    <fx:Declarations>
        <!-- USING A REMOTE SOURCE -->
        <fx:XML id="contactsXML"
            source="http://www.learningflex4.com/xml/contacts.xml"/>
    </fx:Declarations>

    <s:layout>
        <s:VerticalLayout paddingLeft="25"
            horizontalAlign="left" verticalAlign="middle"/>
    </s:layout>

    <s:Label text="Contact Manager" fontWeight="bold" fontSize="14"/>

    <s:HGroup id="managerGroup" gap="25"
        horizontalAlign="left" verticalAlign="middle">

        <mx:DataGrid id="contactsDG" itemClick="contactsDGItemSelect()">
            <mx:columns>
                <mx:DataGridColumn width="150"
                    headerText="First" dataField="firstName"/>
                <mx:DataGridColumn width="150"
                    headerText="Last" dataField="lastName"/>
            </mx:columns>
            <mx:dataProvider>
                <s:XMLListCollection source="{contactsXML.contact}"/>
            </mx:dataProvider>
        </mx:DataGrid>
    </s:HGroup>
</s:WindowedApplication>
```

```

</mx:DataGrid>

<local:ContactDetailComponent id="detailComponent"/>

</s:HGroup>

</s:WindowedApplication>

```

You'll need to create both the main application and its composite component before the application will compile.

Example 16-2. *ContactManager\src\ContactDetailComponent.mxml*

```

<?xml version="1.0" encoding="utf-8"?>
<s:Group
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    height="192" width="324"
    currentState="viewer">

    <fx:Script>
        <![CDATA[
            import mx.controls.Alert;
            import mx.events.ValidationResultEvent;
            import mx.validators.Validator;

            [Bindable] private var isNew:Boolean;
            [Bindable] public var contactID:String = "";
            [Bindable] public var firstName:String = "";
            [Bindable] public var lastName:String = "";
            [Bindable] public var type:String = "person";
            [Bindable] public var email:String = "";
            [Bindable] public var phoneNum:String = "";

            private function onNew():void{
                isNew = true;
                contactID = "";
                firstName = "";
                lastName = "";
                type = "person";
                email = "";
                phoneNum = "";
                this.currentState = "editor";
            }

            private function onEdit():void{
                if (this.currentState == "viewer"){
                    isNew = false;
                    this.currentState = "editor";
                }else{
                    if (validate() == true){
                        if (isNew == true) {
                            Alert.show("This is a new record.");
                        }else{
                            Alert.show("This is an update.");
                        }
                    }
                    // submit to database
                    submitComplete();
                }
            }
        ]]>
    

```

NOTE

It's "dirty" to use so many **Bindable** variables to easily move values between the application and its component, but we chose a path of convenience. A wiser approach would handle this task using several **public** functions called getters and setters that are called to either read or assign values to class variables. Read more about this approach in the sidebar titled "Getters and Setters" on page 357.

NOTE

When **onNew()** is called, the **isNew** variable is set to **true**, the bindable variables are reset with the **type** variable defaulting to **person**, and the state of the component is changed to **editor**.

NOTE

The **onEdit()** function will receive the majority of the changes necessary to integrate a PHP/MySQL solution. Specifically, we'll replace the two **Alert** messages with the code necessary to call **insertContact.php** and the **updateContact.php** scripts, respectively.

```

    }else{
        // invalid data
        // stay in editor state
    }
}
}

```

NOTE

The `submitComplete()` function returns the component to the **viewer** state and calls the main application to reload the XML, as handled by its `refreshApp()` function.

NOTE

Unlike the similar function presented in Chapter 10, the `validate()` function we use here is called by the `onEdit()` function to return a Boolean value confirming whether the input is valid. If the data is invalid, we're still using a **For..Each..In** loop to present error messages.

```

private function submitComplete(event:Event = null):void{
    this.currentState = "viewer";
    this.parentApplication.refreshApp();
}

private function validate():Boolean{
    var isValid:Boolean = true;
    var validatorsArray:Array;
    var errorsArray:Array;
    var errorString:String = "";

    validatorsArray = [firstNameValidator, lastNameValidator,
        emailValidator, phoneValidator];

    errorsArray = Validator.validateAll(validatorsArray);

    if(errorsArray.length > 0){
        isValid = false;
        for each (var error:ValidationResultEvent in errorsArray){
            var formItem:FormItem =
                FormItem(error.target.source.parent);
            errorString += formItem.label;
            errorString += ": " + error.message + "\n\n";
        }
        Alert.show(errorString,
            "There are problems with your submission");
    }
    return isValid;
}

]]>
</fx:Script>

<s:states>
    <s:State name="viewer"/>
    <s:State name="editor"/>
</s:states>

<s:transitions>
    <s:Transition fromState="viewer" toState="editor">
        <s:Fade target="{contactForm}" duration="1000"/>
    </s:Transition>
    <s:Transition fromState="editor" toState="viewer">
        <s:Fade target="{contactGroup}" duration="1000"/>
    </s:Transition>
</s:transitions>

```



```

<fx:Declarations>

    <mx:StringValidator id="firstNameValidator"
        source="{firstNameTI}" property="text"/>

    <mx:StringValidator id="lastNameValidator"
        source="{lastNameTI}" property="text" required="false"
        minLength="2"/>

    <mx:EmailValidator id="emailValidator"
        source="{emailTI}" property="text"/>

    <mx:PhoneNumberValidator id="phoneValidator"
        source="{phoneTI}" property="text" required="false"/>

    <mx:PhoneFormatter id="phoneFormatter" formatString="###.###.####"/>

</fx:Declarations>

<mx:Form id="contactForm" width="324"
    horizontalCenter="0" verticalCenter="0" includeIn="editor">

    <mx:FormItem label="First Name" required="true">
        <s:TextInput id="firstNameTI" width="188" text="{firstName}"
            restrict="a-z A-Z ' \-"/>
    </mx:FormItem>
    <mx:FormItem label="Last Name">
        <s:TextInput id="lastNameTI" width="188" text="{lastName}"
            restrict="a-z A-Z ' \-"/>
    </mx:FormItem>
    <mx:FormItem label="Is a Business?">
        <s:CheckBox id="companyCheckBox"
            label="(check if contact is a business)"
            selected="{(type == 'business') ? true : false}"/>
    </mx:FormItem>
    <mx:FormItem label="Email" required="true">
        <s:TextInput id="emailTI" width="188" text="{email}"
            restrict="^;"/>
    </mx:FormItem>
    <mx:FormItem label="Phone">
        <s:TextInput id="phoneTI" width="188" text="{phoneNum}"
            restrict="0-9" maxChars="10"/>
    </mx:FormItem>
    <mx:FormItem label="">
        <s:Button label="Submit" click="onEdit()"/>
    </mx:FormItem>

</mx:Form>

<s:VGroup id="contactGroup" gap="10"
    horizontalCenter="0" verticalCenter="0" includeIn="viewer">

    <s:HGroup>
        <s:Label text="Name:" fontWeight="bold"/>
        <s:Label id="nameLabel" text="{firstName} {lastName}"/>
    </s:HGroup>

```

NOTE

We removed the custom errors from the validator components for this demonstration. The default errors should be satisfactory.

NOTE

When the **ContactDetailsComponent** is in the **editor** state, the form controls get their values from the corresponding **Bindable** variables, which are set when an item in the application's **DataGrid** is selected.

Note that we're using a special approach called an inline conditional to set the value of the **CheckBox** control. For more information regarding the inline conditional, see the sidebar titled "The Inline Conditional" on page 353.

```

<s:HGroup>
  <s:Label text="Type:" fontWeight="bold"/>
  <s:Label id="typeLabel" text="{type}"/>
</s:HGroup>

<s:HGroup>
  <s:Label text="Email:" fontWeight="bold"/>
  <s:Label id="emailLabel" text="{email}"/>
</s:HGroup>

<s:HGroup>
  <s:Label text="Phone Number:" fontWeight="bold" />
  <s:Label id="phoneNumLabel" text="{phoneFormatter.
    format(phoneNum)}"/>
</s:HGroup>

<s:HGroup>
  <s:Button label="Edit" click="onEdit()"/>
  <s:Button label="New" click="onNew()"/>
</s:HGroup>

</s:VGroup>

</s:Group>

```

The Inline Conditional

The **ContactDetailComponent** uses the following format to define the **selected** value of the detail component's **CheckBox** control:

```
selected="{ (type == 'business') ? true : false }"
```

This syntax is called the *conditional operator*, the *ternary operator*, the *shorthand conditional*, or even the *inline conditional*. You *can't* use a formal **If..Then..Else** statement inside a binding expression, but you *can* use the inline conditional.

As always, the binding expression should occur between curly braces. Within the braces, a set of parentheses defines the condition to be tested; here we're testing whether the value of **type** equals (**==**) the string "business" (the alternate option is "person"). If the condition returns **true**, the value following the question mark (?) is assigned to the binding; if the expression returns **false**, the value following the colon (:) is assigned to the binding.

In the example, it's a coincidence that we're assigning values of **true** or **false** to the **selected** property of the **CheckBox**, either checking or unchecking it. For instance, suppose we wanted to use the **type** variable to determine which function a **Button** should call from its **click** event. If this were the case, we might use the following inline conditional to call the proper function:

```
click="{ (type == 'business') ? addBusines() : addPerson() }"
```

You can see how the inline conditional gives you some extra flexibility when it comes to creating binding expressions.

If you have created both the application and the `ContactDetailComponent`, the initial code base for the `ContactManager` application should be ready to compile.

Go ahead and run the application to get a feel for what you're starting with. It should resemble Figures 16-3 and 16-4, which depict the `ContactManager` in both its **viewer** and its **editor** states.

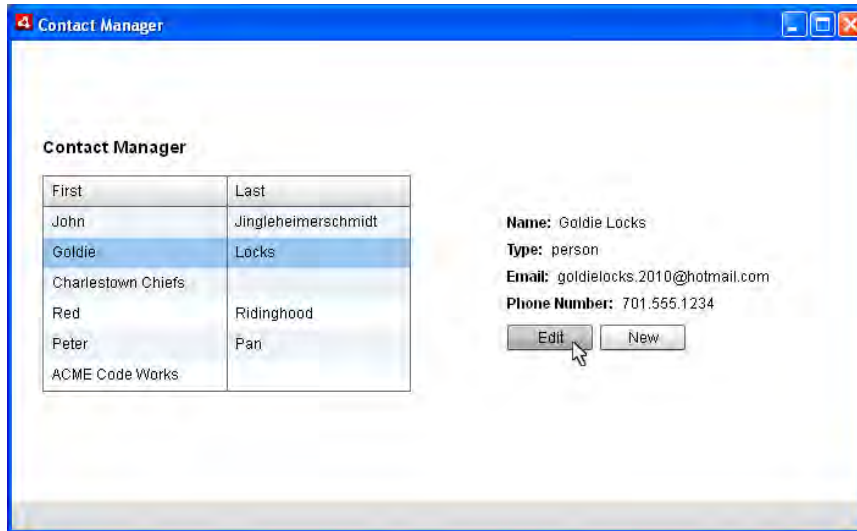


Figure 16-3. The `ContactManager` in the “viewer” state

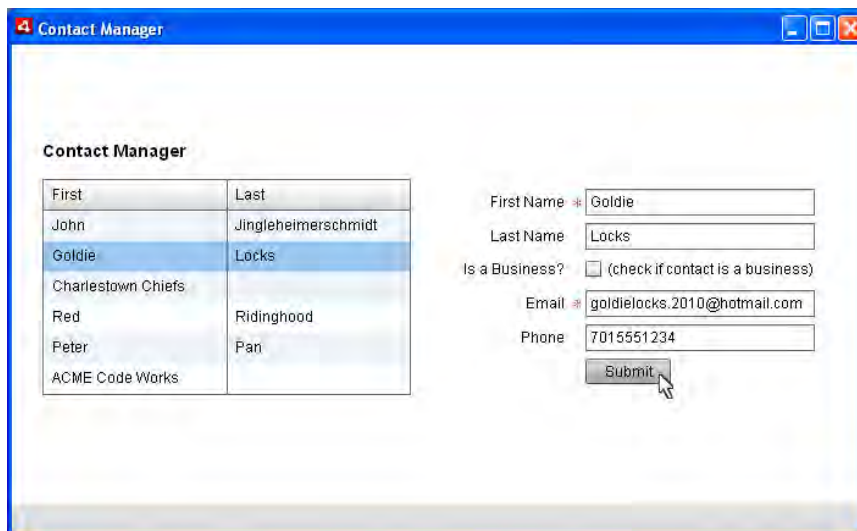


Figure 16-4. The `ContactManager` in the “editor” state

Presently, all you can do is switch the application between its **viewer** and **editor** states by clicking the Edit/New and Submit buttons. Attempting to submit data will only produce an **Alert** message indicating which functionality you attempted. We're about to change that, though.

In the second half of this lesson, we integrate the MySQL and PHP services layer to link the **ContactManager** with our local development environment.

Linking ContactManager to the Server Using the HTTPService Class

In the next phase, we modify the **ContactManager** to replace the XML file with an **HTTPService** call to a PHP script that will request every contact record in a MySQL database and return it in XML format. We also use an **HTTPService** class to call PHP scripts that will either insert new contact records into the database or update existing contact records.

As it happens, there are two separate classes for the **HTTPService** component, one designed for use as an MXML component (`mx.rpc.http.mxml.HTTPService`), and the other designed for use in ActionScript (`mx.rpc.http.HTTPService`). Of course, we will show you both.

Loading Contacts Using the HTTPService Component

We're ready to integrate the first PHP script, but first we have to create it. Start by using whatever method you prefer to add a *php* folder under the project's *src* directory. Next, select the new *php* directory, open the File menu, and chose New→File. In the dialog, provide the filename *loadContacts.php*; be sure to include the file extension.

Getters and Setters

When you create a custom component or a custom class, it's better to use a combination of **getter** and **setter** functions to read or assign values to a class's property variables than it is to use many public **Bindable** variables. Using getters and setters will save your compiled application from becoming bloated by unnecessary behind-the-scenes binding handlers. Also, if many variables are involved, several getters and setters will net performance improvements over the bindings.

Building upon what you learned back in Chapter 5, the following class could be used to store values for **_username** and **_screenName** variables. Notice that the variable names are prepended with an underscore (**_**); this is the recommended convention. Meanwhile, use of the **get** and **set** keywords within the public function definitions determines whether we're accessing a variable to read or to assign its value. Here's the code for an ActionScript class that would belong in a Flex project's default package:

```
package
{
    public class UserClass
    {
        private var _username:String = "guest";
        private var _screenName:String = "Guest";

        public function UserClass()
        {
        }

        public function get username():String{
            return _username;
        }

        public function set
        username(value:String):void{
            _username = value;
        }

        public function get screenName():String{
            return _screenName;
        }

        public function set
        screenName(value:String):void{
            _screenName = value;
        }
    }
}
```

The getter functions are returning strongly typed data, **String** data in this case. On the other hand, the setter functions are returning nothing—the **void** type. Within each getter/setter, we're either returning or assigning a value to a variable of the class. Use of a parameter named **value** in setter functions is pure convention, and you'll see it in other languages as well.

If you create a simple application to test this class, you'll find code completion doesn't suggest both the **public** getter and the setter functions. Instead, you'll be offered a single option for each pair, **username** or **screenName**, much like when you're accessing **public** variables. We're essentially creating properties of the class using a formal approach.

To test the functions in **UserClass**, start a new project, create **UserClass** (an ActionScript class) in the project's default package, and then add the following application code:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx"
    creationComplete="getUserValues()">

    <fx:Script>
    <![CDATA[
        private var userClass:UserClass = new
        UserClass();

        private function getUserValues():void{
            userOut.text = userClass.username;
            screenOut.text = userClass.screenName;
        }

        private function setUserValues():void{
            userClass.username = userIn.text;
            userClass.screenName = screenIn.text;
            getUserValues();
        }
    ]]>
    </fx:Script>

    <s:HGroup horizontalCenter="0" verticalCenter="0">

        <s:VGroup>
            <s:TextInput id="userIn"/>
            <s:TextInput id="screenIn"/>
            <s:Button label="Change Values"
                click="setUserValues()"/>
        </s:VGroup>

        <s:VGroup>
            <s:TextInput id="userOut"/>
            <s:TextInput id="screenOut"/>
        </s:VGroup>

    </s:HGroup>

</s:Application>
```

NOTE

If either the PHP code in Example 16-3 or the MySQL schema defined by Table 16-1 are vague or unfamiliar, please consult the appropriate appendix for either topic. Appendix B presents “MySQL Basics,” and Appendix C presents “PHP Basics.” As we’re only using PHP to bridge our Flex application to the MySQL database (it’s way more powerful than that!), you’ll want to create the database by following Appendix B before considering PHP in Appendix C.

When you emerge into the editor, enter the code in Example 16-3 into the PHP file.

Example 16-3. *loadContacts.php*

```
<?php
header("content-type: text/xml");

// development link:
$link = mysql_connect("127.0.0.1", "root", "");
mysql_select_db("contacts", $link);

// query statement
$query = "SELECT * FROM contact";
$result = mysql_query($query, $link);

$xmlData = "";
$xmlData = "<contacts>\n";
while($row = mysql_fetch_array($result))
{
    $xmlData .= " <contact id='" . $row['id'] . "' type='" .
    $row['type'] . "'>\n";
    $xmlData .= " <firstName>" . $row['first'] . "</firstName>\n";
    $xmlData .= " <lastName>" . $row['last'] . "</lastName>\n";
    $xmlData .= " <email>" . $row['email'] . "</email>\n";
    $xmlData .= " <phone>" . $row['phone'] . "</phone>\n";
    $xmlData .= " </contact>\n";
}
$xmlData .= "</contacts>\n";

// return the XML string
echo $xmlData;
?>
```

This PHP code makes several assumptions about your development environment and your MySQL database. First, it assumes a MySQL server resides on the *localhost* IP, 127.0.0.1, that the **root** username is valid, and that the root password is still an empty string, ""—all the traits of a default MySQL installation. It further assumes there is a database called **contacts**, with a table called **contact**, and that the **contact** table has the schema outlined in Table 16-1.

Table 16-1. *Schema for the contact table*

Field name	Data type	Notes
id	INT	Primary key, auto-incrementing
type	VARCHAR (10)	Either “contact” or “business”
first	VARCHAR (25)	First name value
last	VARCHAR (25)	Last name value
email	VARCHAR (75)	Email value
phone	VARCHAR (10)	Phone value

Considering *loadContacts.php* briefly from a zoomed-out perspective, the script simply queries the **contacts** database for every record in the **contact** table; it then takes the result of the query and concatenates a **String** variable (**\$xmlData**), which it builds according to XML syntax. Together with the header definition, **\$xmlData** should return any records in the **contact** table using the same structure as the *contacts.xml* file we started with.

With the PHP script ready to go and the **contacts** database populated with a few records, return to the **ContactManager** application file and modify the **Declarations** section to include the **HTTPService** as shown in Example 16-4. Also notice that we removed the **source** property from the XML declaration.

Example 16-4. Updating the *Declarations* section of the *ContactManager*

```
<fx:Declarations>

    <s:HTTPService id="phpService"
        url="http://localhost/ContactManager/php/loadContacts.php"
        resultFormat="xml" result="onLoadContacts()"/>

    <fx:XML id="contactsXML"/>

</fx:Declarations>
```

This is very similar to the **HTTPService** component we used in the **YahooSearch** application, only we're pointing it at *loadContacts.php*, and we're taking advantage of its **result** event to call the function **onLoadContacts()**.

Next, Example 16-5 presents a modification to the **refreshApp()** function as well as the new **onLoadContacts()** function. In the **refreshApp()** function, strikethrough styling indicates code that was moved into **onLoadContacts()**.

Example 16-5. Updating the *Script/CDATA* section of the *ContactManager*

```
public function refreshApp():void{
    contactsDG.selectedIndex = 0;
    contactsDGItemSelect();
    phpService.send();
}

private function onLoadContacts():void{
    contactsXML = XML(phpService.lastResult);
    contactsDG.selectedIndex = 0;
    contactsDGItemSelect();
}
```

Since the application's **creationComplete()** event is already calling **refreshApp()**, we changed **refreshApp()** to call the **HTTPService** component's **send()** method. Further, since we're using the service component's result handler to call **onLoadContacts()**, we use that function to set the existing **contactsXML** object's **source** to the **lastResult** of the **HTTPService**, which is returned as XML-formatted data. Do notice that because the **lastResult** is typed as an **Object**, we're casting it as XML using the XML constructor function, **XML()**.

NOTE

You might want to set up a **fault** event handler for the **HTTPService**. If so, you can return to Chapter 11 and review the sidebar titled “It’s Not Your Fault” on page 227. Alternatively, you can inspect how we set up the fault handler in the next section.

At this point, you can launch the application and test the integration. As long as you have some contacts entered into your MySQL database, and as long as your server environment is online, the application should load contacts when you run it.

Sending Data with the HTTPService Class

Now that we’re using the **HTTPService** component to pull data into **ContactManager** via PHP, let’s set up a solution that sends data out of **ContactManager** and into PHP using the HTTP Request type **POST**. We consider the HTTP Request type more seriously later in the sidebar titled “HTTP Request Methods” on page 363.

Let’s quickly discuss *insertContact.php* and *updateContact.php*, and then we’ll integrate them into Flex using an ActionScript-only implementation of the **HTTPService** class. As you’re about to see, the next two PHP scripts are short and sweet, having only the purpose of receiving POST variables from the Flex **HTTPService**, then passing those variables into either a MySQL **INSERT** or a MySQL **UPDATE** query.

Using the same approach you used for *loadContacts.php*, go ahead and create both the *insertContact.php* and *updateContact.php* scripts. Example 16-6 presents code for *insertContact.php*, and Example 16-7 presents code for *updateContact.php*.

Example 16-6. *insertContact.php*

```
<?php
// development link:
$link = mysql_connect("127.0.0.1", "root", "");
mysql_select_db("contacts", $link);

// incoming variables
$type = $_POST['type'];
$first = $_POST['first'];
$last = $_POST['last'];
$email = $_POST['email'];
$phone = $_POST['phone'];

// query statement
mysql_query
(
    "INSERT INTO contact
    (
        type,
        first,
        last,
        email,
        phone
    )
```

```
VALUES
(
    "" . $type . "",
    "" . $first . "",
    "" . $last . "",
    "" . $email . "",
    "" . $phone . ""
),
$link
);
?>
```

Notice in Example 16-6 that the **INSERT** script does not include an **id** variable. That's because the database is automatically adding the **id**, which is the table's auto-incrementing primary key, every time a new record is created.

Example 16-7. *updateContact.php*

```
<?php
// development link:
$link = mysql_connect("127.0.0.1", "root", "");
mysql_select_db("contacts", $link);

// incoming variables
$id = $_POST['id'];
$type = $_POST['type'];
$first = $_POST['first'];
$last = $_POST['last'];
$email = $_POST['email'];
$phone = $_POST['phone'];

// query statement
mysql_query
(
    "UPDATE contact SET " .
    " type = " . $type .
    ", first = " . $first .
    ", last = " . $last .
    ", email = " . $email .
    ", phone = " . $phone .
    " WHERE id = " . $id
    , $link
);
?>
```

In the **UPDATE** statement, notice that we're using an existing record's **id**, its primary key, inside an SQL **WHERE** clause to isolate a single contact record for updates.

As for the Flex integration, Example 16-8 presents changes to the detail component's Script/CDATA block. The **onEdit()** function will take most of the new code, and all of it belongs within the second **if** block following the data validation call. To help clarify the new code within the **onEdit()** function, note the two comments indicating the start and the end of the code addition. The **onFault()** function is entirely new.

NOTE

Every time we perform an **UPDATE**, we're submitting several variables that might not have changed. For example, if we update somebody's phone number, we're also updating his type, first name, last name, and email. You could improve upon this by using some strategic **if** statements to first check and see which values have changed.

Example 16-8. *Modifications to the Script/CDATA block of ContactDetailComponent*

```

private function onEdit():void{
    if (this.currentState == "viewer"){
        isNew = false;
        this.currentState = "editor";
    }else{
        if (validate() == true){

            // additions to onEdit() start here....
            var phpService:HTTPService = new HTTPService();
            var variables:Object = new Object();

            if (isNew == true) {
                phpService.url =
                    "http://localhost/ContactManager/php/insertContact.php";
            }else{
                phpService.url =
                    "http://localhost/ContactManager/php/updateContact.php";
            }

            // we only need to attach the contact id if we're editing..
            if (isNew == false){
                variables.id = contactID;
            }

            if (companyCheckBox.selected){
                variables.type = "business";
            }else{
                variables.type = "person";
            }

            variables.first = firstNameTI.text;
            variables.last = lastNameTI.text;
            variables.email = emailTI.text;
            variables.phone = phoneTI.text;

            phpService.addEventListener(ResultEvent.RESULT, submitComplete);
            phpService.addEventListener(FaultEvent.FAULT, onFault);

            phpService.method = "POST";
            phpService.send(variables);
            // additions to onEdit() end here....

        }else{
            // invalid data.
            // stay in edit mode, don't submit..
        }
    }
}

private function onFault(event:FaultEvent):void{
    var errorString:String = event.fault.content.toString();
    Alert.show(errorString, "Error Submitting");
}

```


By looking through the code adjustment, you can see the `HTTPService` instance can call either the insert or the update script depending on the value of the `isNew` Boolean variable. Here, the `isNew` value decides which script is assigned to the `url` property of the `HTTPService` instance.

The `variables` variable, which is typed as an `Object`, will be used to store so-called *name-value pairs* as `POST` variables for the PHP scripts.

In the code following the `url` property assignment, there are several examples of the `variables` instance receiving new name-value pair assignments. Ultimately, each contact variable is assigned to `variables`; the `id` variable is the only exception to this, as it's attached only when we're *not* submitting a new record.

Following the name-value pair assignments, we add two event listeners. The first event listener responds to the `result` event and calls the existing function `submitComplete()`, which changes the state of the detail component back to `viewer` and calls `refreshApp()` on the main application to refresh the service component's `lastResult` and subsequently reload the `DataGrid` with the most recent data. The second event listener responds to the `fault` event and calls the `onFault()` function, which creates an `Alert` message that will state the details of the fault.

Finally, the `HTTPService` component's `method` property is set to the string value `"POST"`, and then its `send()` method is called, which passes the `variables` object in as the data payload consisting of several name-value pairs.

See Figures 16-5 and 16-6 for screenshots of the finished AIR application.

HTTP Request Methods

HTTP Requests can take the form of several methods: **GET**, **POST**, **HEAD**, **OPTIONS**, **PUT**, **TRACE**, and **DELETE**. By and large, **GET** and **POST** are the most frequently used request types, and for the Flex `HTTPService` class, **GET** is the default.

The **GET** method might seem familiar, as it sends request variables directly through the URL. If we had used the **GET** method for these examples, we might have been dealing with `url` values that looked comparable to this: <http://localhost/ContactManager/loadContacts.php?id=1&type=person>.

The **POST** method, however, is used to submit data more discreetly, embedded within the URL request. However, don't misconstrue this as meaning untouchable, secret, or, especially, *secure*. **POST** requests might not be obvious, but they can be intercepted.

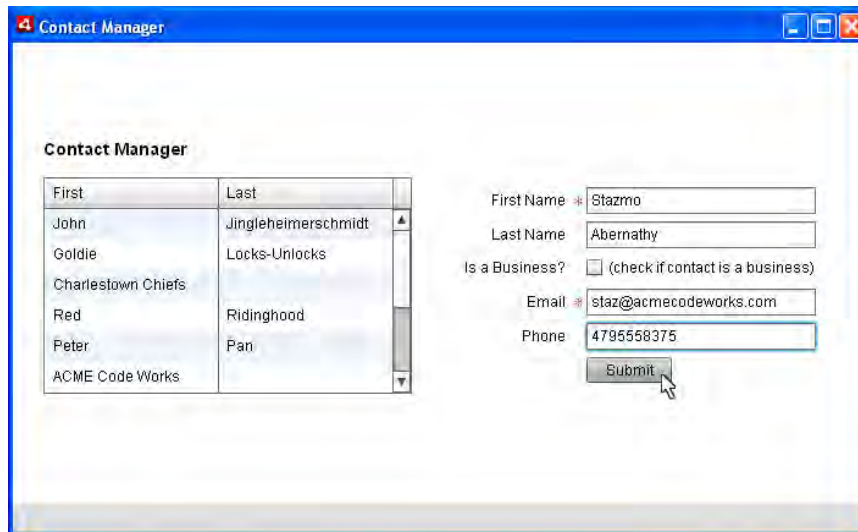


Figure 16-5. Creating a new record for the contact table

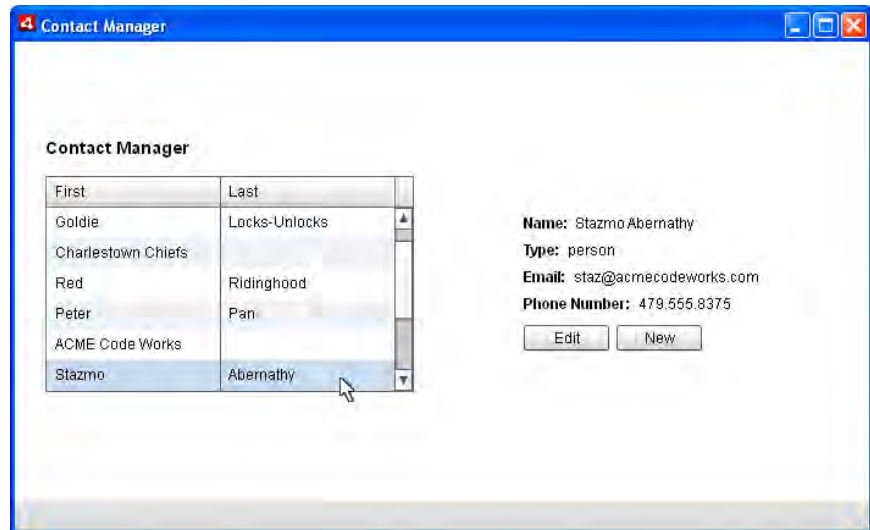


Figure 16-6. Viewing the newly saved record

Summary

This chapter presented you with a big-picture example demonstrating how to integrate Flex applications with a PHP/MySQL server solution using the **HTTPService** component (MXML) and its ActionScript counterpart. Altogether, you learned the following:

- Creating a desktop application in Flex is as simple as creating a new project that runs in Adobe AIR.
- You can link Flash Builder to a local development environment by defining the server and identifying an Output folder within the server's web scope.
- Depending on your needs, you can invoke the **HTTPService** as either an MXML component or an ActionScript class.
- You can use PHP scripts to request data for Flex, and via the HTTP Request method **POST**, you can also submit Flex data as an **Object** of name-value pairs into a PHP script and relay them to a database.
- The ternary operator can be used within MXML property bindings when you need a simple inline condition to determine an attribute's value assignment.
- Getter and setter functions are preferable to many bindable variables, as too many bindable variables result in larger compiled application sizes and can lead to degraded performance.

In Chapter 17, we demonstrate how to deploy your applications to both the Web and the desktop. Specifically, we discuss how to deploy the **PhotoGallery** as a web-based Flex application. We also demonstrate how to create an AIR installer, which you can use to deploy the **ContactManager** to the desktop as an Adobe AIR application.

DEPLOYING FLEX APPLICATIONS

In this next-to-last chapter, we cover the processes for deploying Flex applications to either the Web or the desktop.

Deploying to the Web

First we discuss the process of deploying to the Web, and demonstrate the steps relative to the **PhotoGallery** application. A typical web deployment involves the following steps and considerations:

- Check the Flash Builder Compiler options
- Modify the *index.html.template* file
- Merge or separate the Flex Framework Library
- Export a release build
- Upload files to the web host

As you know, we've been compiling our applications to the *bin-debug* directories of their respective projects. The compiled SWF in the *bin-debug* folder contains extra debugging methods used by the debug version of Flash Player. When you casually run a project in order to test it, Flash Builder launches the debug version of your application. Of course, the debug version is necessary for debugging and profiling (we briefly discuss profiling in the sidebar "Tweaking Performance" on page 375), but it's not yet optimized for frequent downloading. Therefore, when you're ready to deploy, you'll want to create a release build that doesn't include the debugging extras. The specific advantage of a release build is its smaller compiled size.

Now we'll walk through the steps identified in the previous list and create a release build for the **PhotoGallery** application, so open that project now.

IN THIS CHAPTER

Deploying to the Web
Deploying to the Desktop
Summary

Checking Compiler Options

Before creating a release build, it's wise to double-check a handful of compiler options and project properties, so we'll start there.

With Flash Builder open and the **PhotoGallery** application loaded, open the Project menu and select Properties. In the property dialog's navigation menu, select Flex Compiler (Figure 17-1).

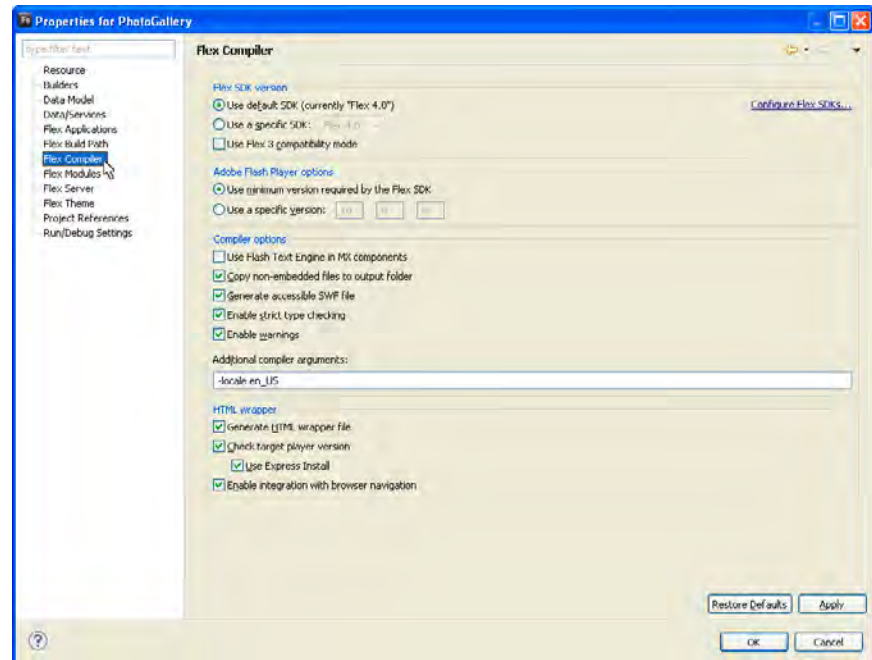


Figure 17-1. Reviewing compiler options in Project→Properties→Flex Compiler

Flex SDK version

The first series of options in the compiler properties dialog allows you to add, remove, or select the Flex SDK version used to compile a project. The Flex 4 SDK is used by default, so you shouldn't need to modify the SDK setting. If another project requires using either an older or newer version of the Flex SDK, select “Configure Flex SDKs...” to access a dialog that allows adding, removing, and selecting the default SDK revision.

Adobe Flash Player options

This setting allows you to manually specify which revision of the Flash Player you require for your projects. Rather than manually specify a preference, it's best to leave the default setting, “Use minimum version required by the Flex SDK”.

Use Flash Text Engine in MX components

Flash Player 10 introduced significant differences to the player's text-rendering architecture, and Spark components are designed to take advantage of the improvements (one of which is the ability to rotate nonembedded fonts). Although older Halo components don't naturally support the new architecture, this option tells the compiler to extend some new text features to the MX components. Note that doing so increases the size of your compiled SWF. Barring some particular circumstances, it's likely best to leave this option unchecked.

Copy non-embedded files to output folder

The setting "Copy non-embedded files to output folder" specifies whether to copy supporting files such as XML, graphic assets, etc., into the bin. Leave this item turned on to copy over files such as *photos.xml*, which the **PhotoGallery** needs to run. If you prefer to manually copy the necessary files, you can disable this item—but beware: any time you change a source file, you'll need to manually copy the source file to the output directory.

Generate accessible SWF file

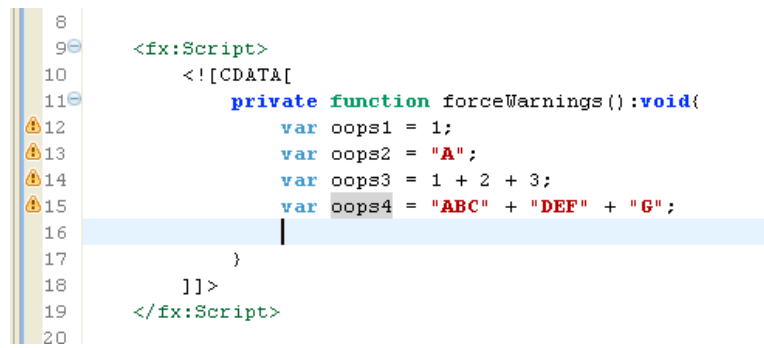
The next item concerns accessibility. The option "Generate accessible SWF file" refers to enabling screen reader support for your application. A screen reader is a special application that recites the screen content to a user with impaired vision. Flash Builder 4 enables this option by default, which does result in a slightly larger SWF, as a bit of extra code is added to the compiled version of your application to enable screen reader support. Unless you never expect your users to require a screen reader—such as when you're developing an application for a specific group of users—it's helpful to enable this option. In spite of this simple setting, however, creating truly accessible applications involves a little more attention to detail. For more information, see the sidebar "Designing Accessible Applications" on page 368.

Enable strict type checking

Strict type checking is another option you should leave enabled, as it forces data type agreements at compile time and provides a subtle improvement to runtime operation.

Enable warnings

Compiler warnings are noticeable alerts that appear in the editor pane when the Flex compiler takes exception to a line of code. For example, if you forget to type a variable, assuming warnings are enabled, Flash Builder attempts to get your attention to fix the code using a better practice. Figure 17-2 provides an illustration of this functionality. We recommend leaving compiler warnings enabled.



```

8
9
10 <fx:Script>
11   <![CDATA[
12       private function forceWarnings():void{
13           var oops1 = 1;
14           var oops2 = "A";
15           var oops3 = 1 + 2 + 3;
16           var oops4 = "ABC" + "DEF" + "G";
17       }
18   ]]>
19 </fx:Script>
20

```

Figure 17-2. Compiler warnings indicating a bad practice that might lead to runtime errors

Designing Accessible Applications

Enabling “Generate accessible SWF file” is the first step toward making your applications more accessible, but it’s not a complete solution.

You might not have thought about it, but many people with challenges or impairments may want to use your application. Besides users with limited or no vision, some might simply have color blindness. For such an individual, it’s important to make sure your application doesn’t rely heavily on color differentiation; similarly, make sure the application’s color schemes don’t interfere with readability. For users with weak vision, make sure your application works at lower screen resolutions. Even blind users should—at the very least—be able to interact with your application enough to know that it’s a photo gallery. Even though blind users will not be able to see the images, they might be interested in descriptions of your images.

Here’s another caveat to ponder: be careful with audio, particularly sound effects. For some users, undesired sounds can conflict with a screen reader, so if you opt to use sounds, provide an easy way to disable them. Conversely, make sure serious audio offers captioning.

Your application should not rely on the mouse as the sole input device. Said differently, your entire application should be accessible via the keyboard. Some limitations might prevent people from using a mouse, but that shouldn’t preclude them from using your application. All standard Flex components offer

keyboard navigation, so usually this is not a problem. Some features of your application might be difficult to control without a mouse; nevertheless, a good rule of thumb is to provide multiple means of accomplishing a task. So, if you manage a particular feature or control using the mouse, provide an obvious alternative using the keyboard.

Remember the **tab index**, the sequential order by which focus changes as the Tab key is pressed? Typically Flex provides a logical tab index sequence based on the arrangement of controls in a layout. However, it’s still wise to test your application’s tab index, especially if you’re using a custom order. One of the best ways to ensure you’ve designed an accessible application is to test it using only a keyboard.

Finally, make certain your application is intuitive and easy to navigate. Provide icons that enhance the application, but don’t rely on them to convey meaning. For instance, we added a magnifying glass icon to the search button in the **YahooSearch** application, but a label on the button still reads “Search”. The icon alone just doesn’t suffice. In a busy application, such an icon might be interpreted as “zoom in.”

Making your applications accessible often makes them cleaner, simpler, and easier to operate for your users, not just those with impairments. So, follow these simple rules every time, and you’ll ensure you’re creating not just accessible applications but well-designed applications.

HTML wrapper

The HTML wrapper settings relate to an HTML file autogenerated for your application by Flash Builder; it includes code to embed *PhotoGallery.swf* in a web page. Even if you've created your own HTML page, it's smart to leave "Generate HTML wrapper file" enabled so you'll have a page that launches when you test and debug. (For information on customizing the generated HTML file, see the sidebar below, "Customizing the HTML Wrapper.")

Customizing the HTML Wrapper

Autogenerating an HTML file may be fine for many developers, but sometimes you'll want to modify the template for the HTML wrapper or even create your own HTML file.

The HTML template is located under your project's *src* folder in a folder called *html-template*. This directory contains additional files that may be copied into your output folder, but the main file is called *index.template.html*. This is not the final HTML file that will be generated; rather, it's a template the Flex compiler uses to create an HTML page that embeds the Flex application.

The template contains variables, or tokens, such as **`${title}`**, which are special placeholders the Flex compiler replaces with real values. Table 17-1 lists the available tokens. The default values of some tokens are generally fine, but you might want to modify some of them, such as **`${title}`**, which is the text that displays as the HTML page's title.

You can also modify other parts of this file, such as the content that displays if a user doesn't have Flash Player installed. You can find this area by looking through the file for **`alternateContent`**.

Table 17-1. HTML wrapper token list

Token name	Description
<code>\${application}</code>	This option sets <code>id</code> and <code>name</code> properties of the embedded SWF file, allowing JavaScript or other browser scripting languages to access it.
<code>\${bgcolor}</code>	This is the background color of the HTML file. A Flex application might not occupy the entire browser window, so this property gives you the privilege to make sure the default background color doesn't interfere with your color scheme.
<code>\${height}</code>	This is the <code>height</code> of the application, set by the <code><s:Application/></code> tag's height property.
<code>\${swf}</code>	This token sets the path to the compiled application's SWF file.
<code>\${title}</code>	This is the title of the HTML page, which displays in the title bar of the browser. By default, this is the name of the Flex application, such as <code>PhotoGallery</code> .
<code>\${version_major}</code>	This is the required major version number of Flash Player, for example, 10. This token appears only in wrappers with Flash Player version detection code, which is enabled in the HTML wrapper compiler options.
<code>\${version_minor}</code>	This is the required minor version number of Flash Player, as set in the HTML wrapper compiler options.
<code>\${version_revision}</code>	This is the required revision version number of Flash Player, as set in the HTML wrapper compiler options.

Flash Player version

The next item in the HTML wrapper category authorizes checking the version of the Flash Player plug-in. We recommend leaving the defaults here. The option “Use Express Install” enables the application to prompt people to update their versions of Flash Player if they don’t have the required version. Turning on this box creates a SWF file called *playerProductInstall.swf*, along with some extra code in the HTML wrapper, in the project’s output directory.

Enable integration with browser navigation

The final item in this dialog box is “Enable integration with browser navigation”. Checking this item outputs supporting files (in the history folder of your project) that synchronize the browser’s history management with your application, such as the ability to use the browser’s Back and Forward buttons. The **PhotoGallery** application doesn’t use history management, so it’s fine to turn off this feature.

Merge or Separate the Flex Framework Library

The next question you’ll face concerns whether to merge the Flex framework library into your application or to separate it as supporting *Runtime Shared Library* (RSL) files. Using RSL linkage between your Flex application and the framework libraries results in a significantly smaller SWF size.

You can access the merged/RSL option by opening the Project menu, selecting Properties, navigating to Flex Build Path, and then opening the Library Path tab (Figure 17-3).

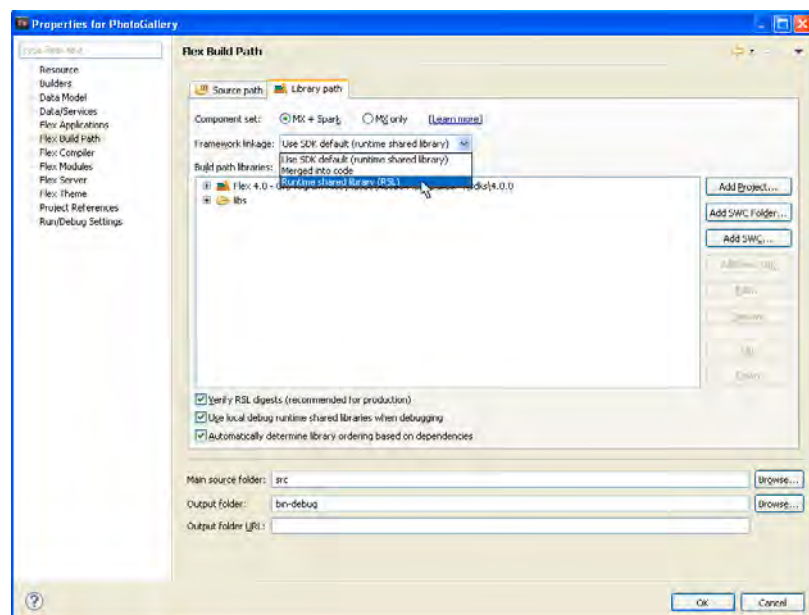


Figure 17-3. Selecting Runtime Shared Library as the library path

RSLs create performance improvement by caching the Flex framework on the local machine so it doesn't have to load every time a Flex application is viewed. Instead, RSLs separate the framework into six separate files, and whenever a Flex application is initially loaded, the framework files are downloaded behind the scenes and cached on the local computer. Then, the next time the user loads that Flex application—or another Flex application leveraging *framework caching*—she won't have to download the framework again, because her machine will default to the cached local files.

In practice this means the first download of an RSL-linked Flex application will be larger and take longer. However, after the initial download, subsequent downloads will be shorter and faster.

The default settings for this option changed between Flex 3 and Flex 4. Flex 3 favored merged libraries, but Flex 4 defaults to RSLs. It's difficult to imagine a web deployment that would not benefit from RSL linkages, which is probably why Adobe made RSLs the default.

Export a Release Build

After reviewing build options for your project, it's time to create the release. You create a release build by selecting the project in the Package Explorer pane, then following Project→Export Release Build. This displays the dialog shown in Figure 17-4.

NOTE

Any Flex application viewed in a web page should be cached by the web browser. Therefore, whether you're using framework caching or not, the application will load immediately upon subsequent visits. However, because frameworks are stored in locations away from the history and temporary Internet files, they will remain intact even when the browser's cache is cleared.

NOTE

*Modules, another Flex feature, allow you to modularize parts of your application into smaller, self-contained pieces. This is a useful approach for larger applications that have many views or parts. If you think modularizing might improve your applications, check the documentation for the **Module** component.*

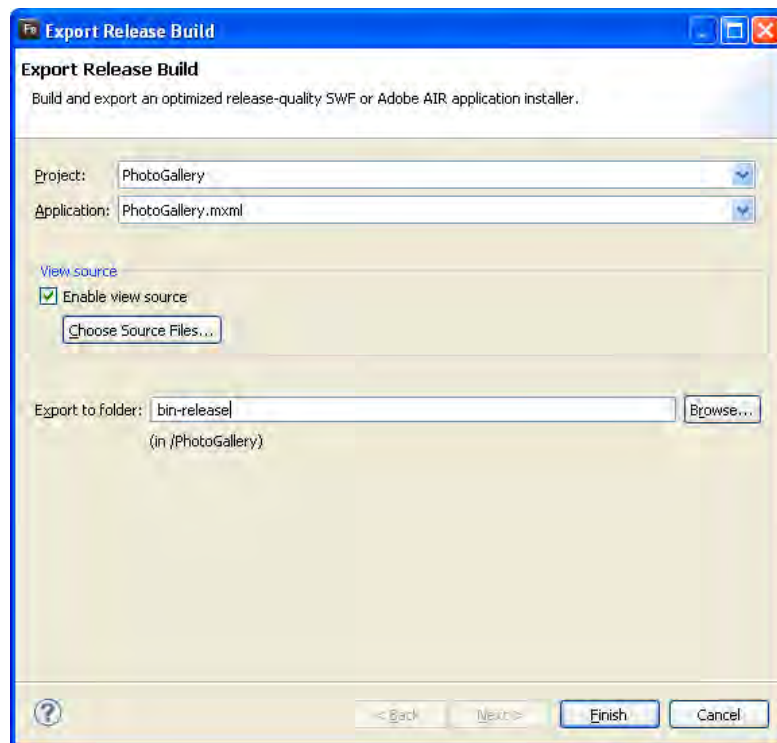


Figure 17-4. The Export Release Build dialog box

Because you selected a project first in the Package Explorer, the Project and Application fields should be prefilled with **PhotoGallery** and *PhotoGallery.mxml*.

Sharing source code

The middle section of the Export Release Build dialog is the “View source” section. Checking “Enable view source” will add a folder to your release directory containing a web page and your project’s source code. The source page will present a list of files that make up your application, as well as a custom viewer that allows users to select and review your code (see Figure 17-7, later in this section). The page also contains a link to download the entire offering as a ZIP file archive.

If you feel comfortable sharing your code, enable this option. Sharing source code is a great way to help others learn Flex. It’s also a way to give tech-savvy users access to the internals of your application so that, if they find problems, they might give you feedback concerning any shortcomings or failures.

If you do enable source, though, you may not want to include every single file. Accordingly, you can choose which files you want to share. To selectively include and exclude files from the source package, click the Choose Source Files button. This opens the Publish Application Source dialog box shown in Figure 17-5, which contains a tree list of your source files. In this case, we included code within the *src* directory, and chose not to include the *html-template* source or the empty *libs* folder. While you’re here, you can even change the name of the directory where your source code will be saved, if you’re inclined.

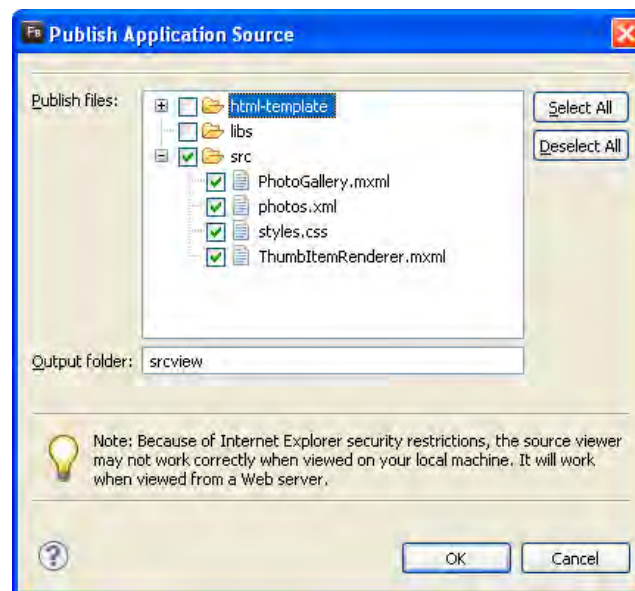


Figure 17-5. The Publish Application Source dialog box

WARNING

Even if you don’t enable view source, you should realize that it’s possible to decompile a Flex application to expose its source code. Therefore, it’s been recommended that “business-sensitive” routines and algorithms be kept out of the Flex application.

If you enable the view source option, the `<s:Application/>` tag of *PhotoGallery*. *mxml* will gain a new attribute—`viewSourceURL`—that points to your new source directory. The attribute will cause your application to create an extra context menu item called View Source, which users of your application will see if they right-click on the application (Figure 17-6). Selecting View Source opens the Source view, as shown in Figure 17-7.

WARNING

If you're developing an application for a company that doesn't want to release its source, certainly disable the view source option for the release build. If you've previously enabled view source, make sure the source files don't remain in the output directory. Even if you disable view source for the application, if you accidentally upload the source files to a web server, they will likely become available.

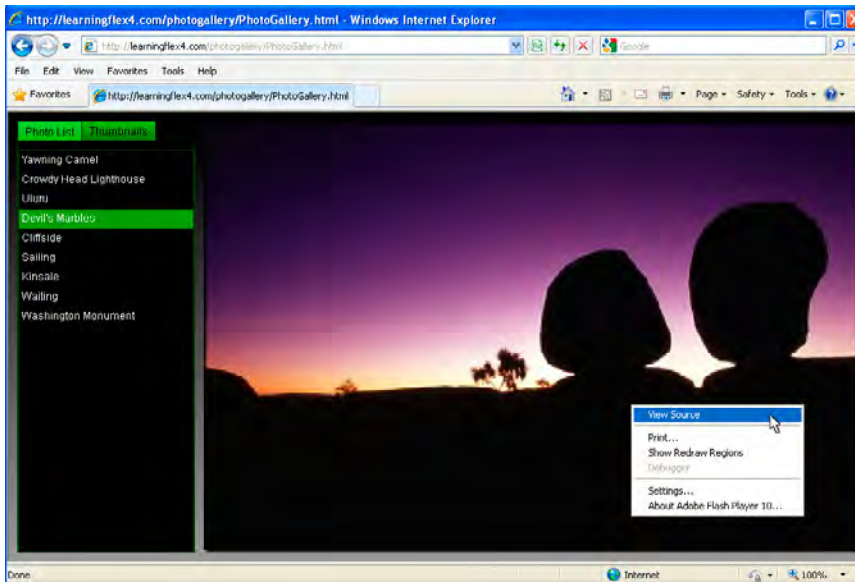


Figure 17-6. The PhotoGallery application, showing the context menu

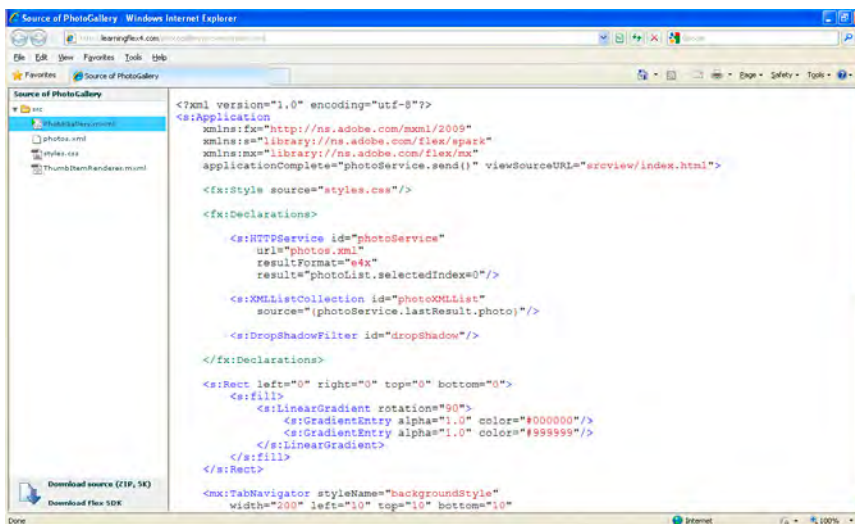


Figure 17-7. Source view for the PhotoGallery application

Setting the export path

The last option in your Export Release Build dialog is “Export to folder,” which lets you establish the location and name of the folder that will contain source code. Unless you prefer a different folder name or want to place the build in a different directory, just accept the default of *bin-release* and click Finish. As shown in Figure 17-8, a new folder named *bin-release* will be added to the **PhotoGallery** project. This folder contains only the files necessary for the application to run in a browser.

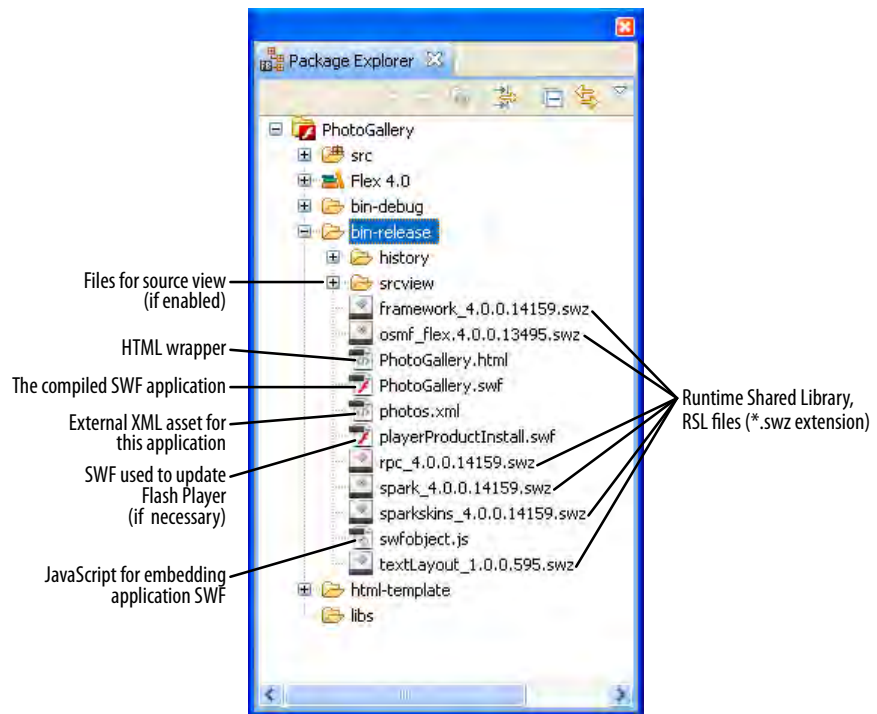


Figure 17-8. Release build folder structure for the PhotoGallery

NOTE

If you're planning to add a server technology behind your Flex application, such as PHP, ASP.NET, ColdFusion, etc., make sure your web hosting candidate supports your chosen technology.

Upload files to the web host

Now that you have a release build, you'll need to upload it to a server to make it available on the Web. If you don't currently pay for web hosting or have a space allotted through your school or workplace, you might need to do some shopping.

Once you have a web server/host, use a File Transfer Protocol (FTP) client to copy the contents of the release build into an appropriate directory on your server. Depending upon the domain name and folder structure of your host, the URL to access your application will change. For an example, though, you can see the final version of the **PhotoGallery** application on the book's

companion website at <http://learningflex4.com/photogallery/PhotoGallery.html>, because we copied the contents of our release build into the **photogallery** directory of the web server, located at learningflex4.com. You can see the final result in Figure 17-9 or by navigating to that URL in your web browser.

NOTE

If you don't already have an FTP client, check out FileZilla, which is quite a nice offering from the open source community: <http://filezilla-project.org/>.

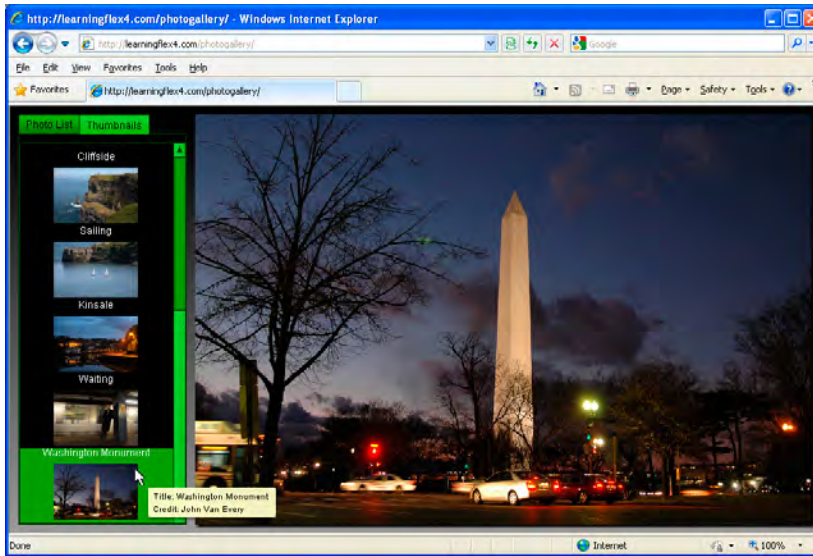


Figure 17-9. The PhotoGallery, deployed as a web application, viewed in a browser

NOTE

If you change the name of the HTML file to `index.html`, most web servers will automatically load that file when navigating to the file's directory. For example, we renamed a copy of our HTML file from `PhotoGallery.html` to `index.html`, and now you can load the application by browsing to <http://learningflex4.com/photogallery/> instead of <http://learningflex4.com/photogallery/PhotoGallery.html>.

Tweaking Performance

Another aspect of software development is performance assessment. Although you should always be aware of performance as you write code, some feel it's wise to wait until core development is concluded before you become too concerned with "tweaking" an application.

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

—Donald Knuth

However, when you're ready to work on performance, one tool at your disposal is Flash Builder's **Profiler**, which is included with the Flash Builder Premium edition. The Profiler is used to investigate an application for performance bottlenecks, and it functions by sampling the time required by different parts of an application to complete their functions. The results allow you to see, at a glance, which functions in an application have the greatest effect on performance. Of course, you use the information you glean to optimize your application.

To see for yourself, open the Profiler by selecting **Run**→**Profile** or by clicking the Profiler toolbar button (Figure 17-10).



Figure 17-10. The Profiler toolbar button

User (or Friend) Testing

The best way to ensure your applications are usable, or user-friendly, is to get comments from others. Watching people interact with your applications can yield invaluable insights into streamlining and perfecting the *user experience* (UX). As the developer, you become so immersed in an application while creating it that poor interaction may start to make perfect sense. Meanwhile, someone who has never seen your application may be completely flummoxed.

If you don't have a UX lab or the budget to establish one, a few friends will work in a pinch. Just hand your application over to a pal or two, then watch. Don't explain how things work or answer any questions—just sit back and watch. Remember, you won't be around to help others, so danqsyle cheat by giving them privileged information.

Listen to their comments and remain patient, even if you don't always agree with them. Chances are, you'll elicit some great feedback that will help you work out kinks.

Deploying to the Desktop

Because the **ContactManager** application relies on PHP and MySQL for data delivery, deploying the full application isn't as easy as double-clicking an icon. However, it shouldn't be daunting, either. Installing an AIR application in a WAMP environment will do just fine for now. Remember, though, if you were to take the concepts from this application and make something more purposeful, and then deploy it for production, the following overall guidelines should be observed:

- WAMP should not be used for a production environment; rather, formal installations of Apache, MySQL, and PHP should be used instead.
- Your PHP scripts should be copied into the web scope of the PHP server.
- References to the MySQL database within the PHP scripts should be checked and updated if necessary.
- A database schema should be established on the MySQL server; if the schema is changed (e.g., field names are changed), the PHP scripts should be updated to accommodate this.
- The PHP scripts called by **ContactManager** are hardcoded in the application; their paths should be checked and updated if necessary, and the application recompiled.

That workflow would get you on track to deploy an AIR application similar to the **ContactManager** in a production environment. To install an AIR application, though, we need to create an installation utility.

Creating an AIR Installer

Creating an Adobe AIR installer is fairly easy. In fact, it's not much more complicated than performing a release build for a web application. However, we're going to add a few steps for good measure.

We can itemize the tasks necessary to make our AIR installation utility as follows:

- Customize the **ContactManager** to include a unique taskbar/dock icon.
- Export an installer file.
- Create a seamless installation utility for web-based distribution.

Customizing the Application

The icon appearing in the taskbar/dock while we run the **ContactManager** is a default AIR icon. We should replace it with a custom icon for deployments, and to do that, we need to hack a couple of lines in *ContactManager-app.xml*, the so-called *AIR application descriptor file*.

The application descriptor file is similar to the *html-template* used in Flex web applications, except this file is a template for adjusting traits of your application. Specifically, the descriptor file lets you adjust qualities such as the words displayed in the window's title bar, the look of the window, and the icon that's displayed in the taskbar or dock.

Because it's filled to the brim with comments, you can easily peruse the descriptor file looking for properties you want to modify. Furthermore, many options are commented out, so you can uncomment these and adjust them as needed.

We'll modify this file in order to replace the default icon the application displays. In this case we'll use a 128 × 128-pixel PNG graphic that Alaric created. Download it from http://learningflex4.com/contactmanager/address_icon.png and save it to the project's *src* folder. Then, find the following section of the descriptor file:

```
<!-- The icon the system uses for the application. For at least one resolution,
      specify the path to a PNG file included in the AIR package. Optional. -->

<!-- <icon>
      <image16x16></image16x16>
      <image32x32></image32x32>
      <image48x48></image48x48>
      <image128x128></image128x128>
</icon> -->
```

Modify it like so:

```
<!-- The icon the system uses for the application. For at least one resolution,
      specify the path to a PNG file included in the AIR package. Optional. -->

<icon>
  <image128x128>address_icon.png</image128x128>
</icon>
```

NOTE

You can set the **height** and **width** of the AIR application in the descriptor file, but you can also set a default height and width using the **height** and **width** properties of the `WindowedApplication`.

Uncommenting the icon tag set, removing the icons types we won't need, and adding the 128 × 128-pixel `address_icon.png` is all we have to change to enable a custom icon. For the best results, you can create your own versions of the 48 × 48, the 32 × 32, and the 16 × 16 icons, but smaller versions of our 128 × 128-pixel icon will be scaled down as needed.

Exporting an Installer

To export an installer for an AIR application, you use the same command as for a web application, `Project`→`Export Release Build`. This opens the Export Release Build dialog, just as if you were exporting a release for a web application (Figure 17-11).

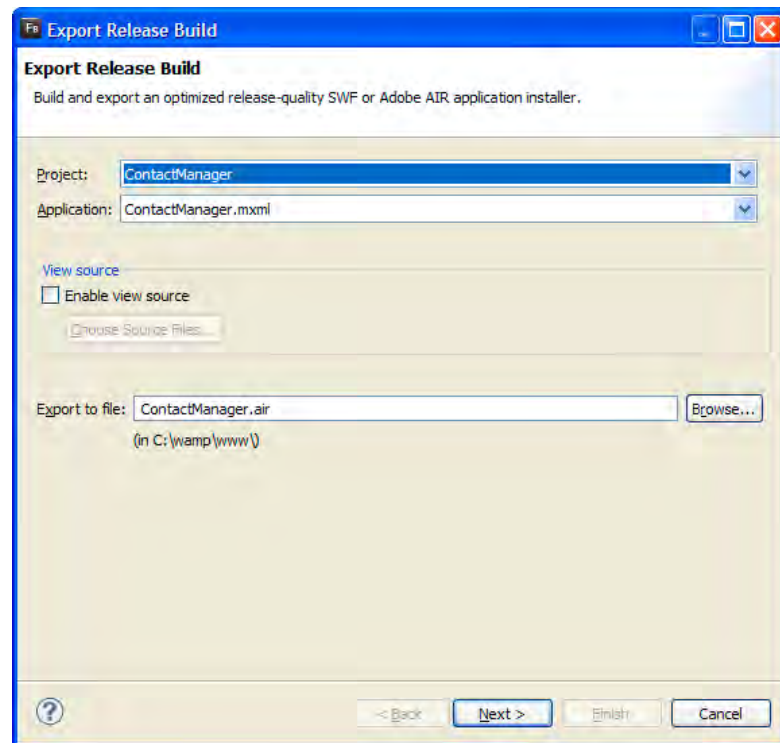


Figure 17-11. Exporting an AIR application: Step 1

However, unlike the release build for a web application, an AIR release requires some additional information. Specifically, an AIR application, because it's installed as a desktop application, requires a *digital signature*. Click `Next` in the Export Release Build dialog box, and you'll see the Digital Signature dialog, as shown in Figure 17-12. A digital certificate is a security measure put in place to make certain that your application comes from you and wasn't modified by anyone else.

For the install to work correctly, you need to select the first option, “Export and sign an AIR file with a digital certificate.” This option allows you to submit a digital certificate you’ve procured formally from one of the various certificate sites; alternatively, it allows you the option of creating your own digital certificate. Unless you’ve gone through the process of obtaining a security certificate, you’ll need to create your own by clicking the Create button.

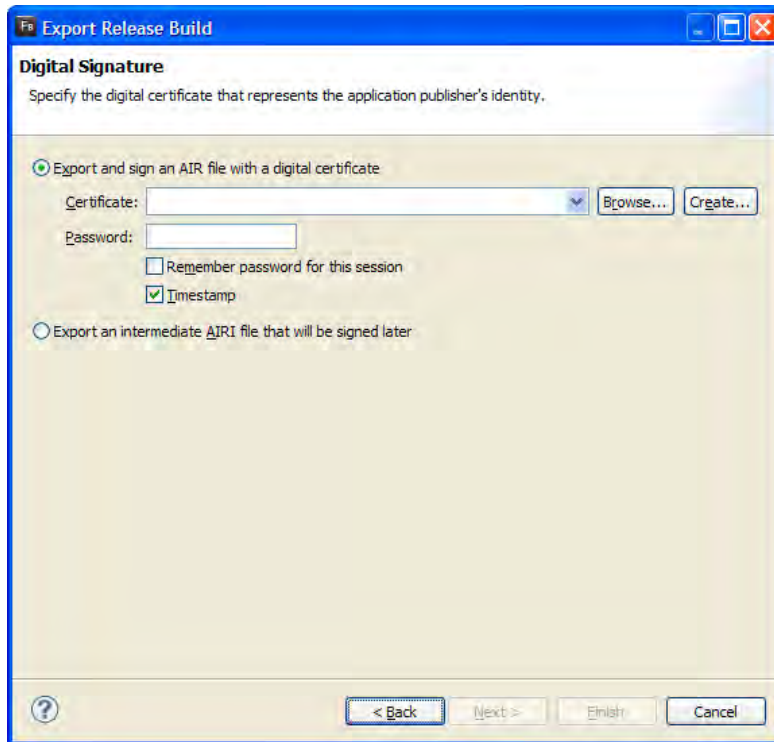


Figure 17-12. Exporting an AIR application: Step 2

About Digital Certificates

A digital certificate provides a means of verifying that a desktop application was not altered since the time it was signed. This certificate is in place as a security measure, and it's also used to verify the application publisher's identity.

When an AIR file is signed with a “self-signed” digital certificate, the publisher information can't be verified. Adobe AIR can tell that the installation has not been altered since it was signed, but it has no way of proving the identity of the publisher who signed the file. Because of this, the publisher will be displayed as “UNKNOWN” in the installation dialog box, as shown in Figure 17-19.

To learn about getting a certificate from one of the trusted authorities, check the following websites:

- VeriSign: www.verisign.com
- Thawte: www.thawte.com
- Microsoft Authenticode: <http://msdn2.microsoft.com/en-us/library/ms537364.aspx>

This prompts you with the dialog box shown in Figure 17-13, which lets you create a certificate for your application. All that is required is a publisher name and a password, as well as the certificate name. In this case, we named the certificate **contactmanager.certificate**, though the name doesn't really matter. Be sure to remember this password, because you'll be required to enter it any time you use the certificate you're creating.

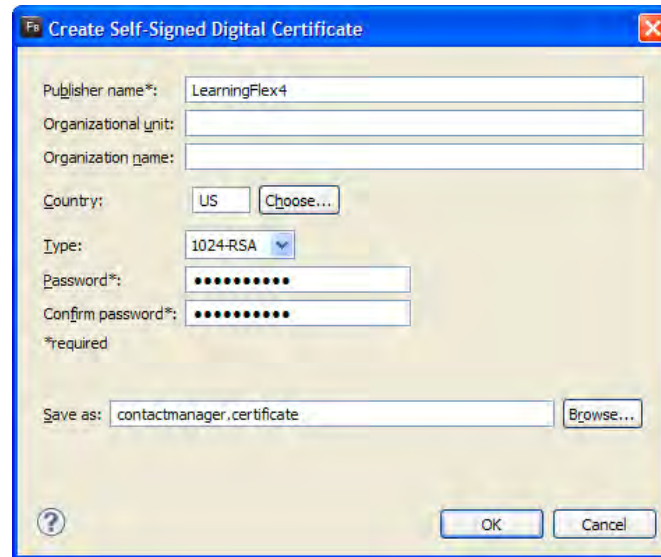


Figure 17-13. Exporting an AIR application: Step 3

Once you create your certificate, you'll return to the Export Release Build dialog box, where you'll be prompted to enter the password for your certificate again (Figure 17-14). Once you've entered it, continue to the last step of the process.

The final step in creating an installer for your application is specifying the files you want to include (Figure 17-15). Ensure that you include the required files, but nothing extraneous; for instance, we're not including the recently phased-out *contacts.xml*. The compiled SWF and the application descriptor file are always required because these two files are necessary for the application to run. You'll also need to include *address_icon.png* or your own custom icon. If you chose to include source code, a directory for the code will also be included (typically *srcview*).

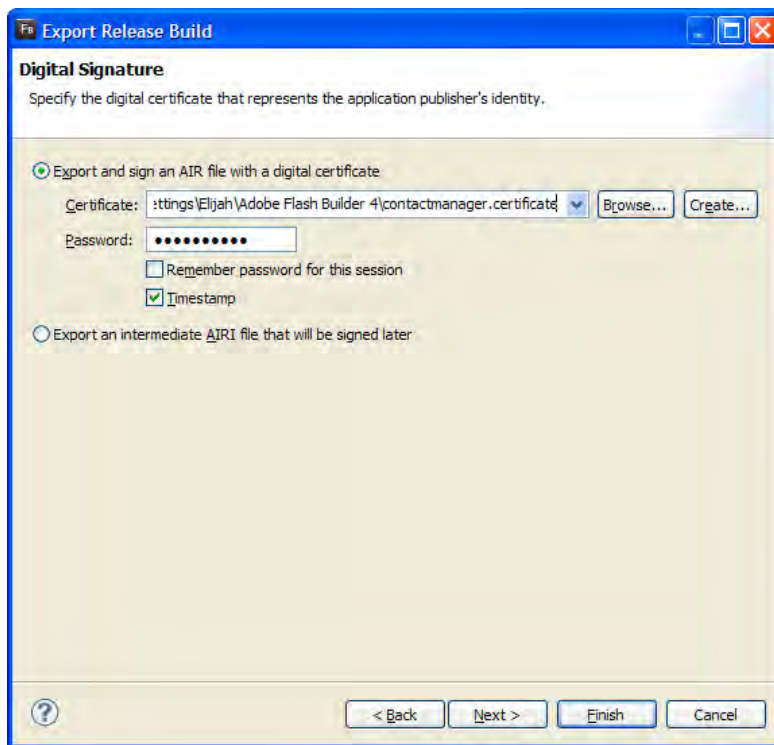


Figure 17-14. Exporting an AIR application: Step 4

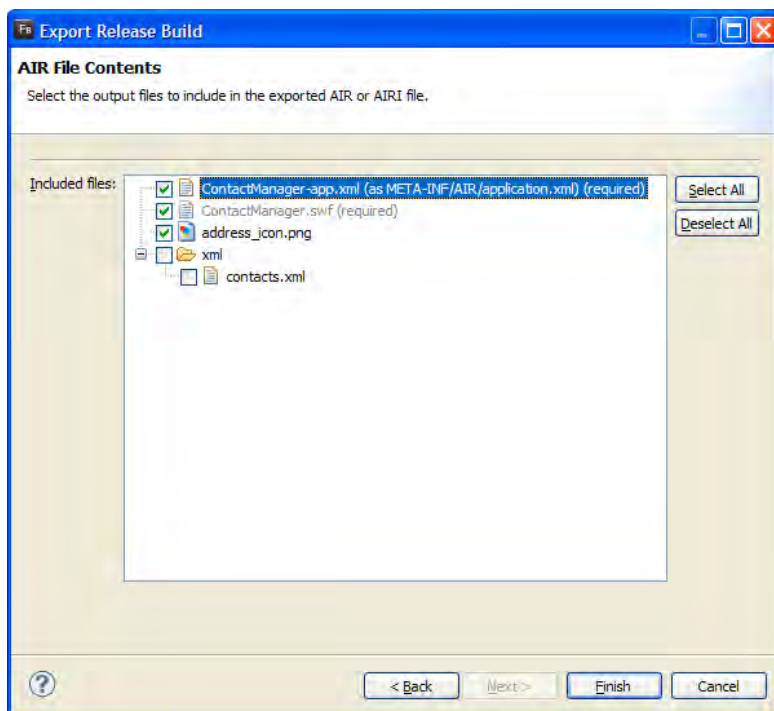


Figure 17-15. Exporting an AIR application: Step 5

Once you click Finish, a file called *ContactManager.air* will be created in the top level of the project. This is your installation file for distributing your application, similar to a DMG file for a Mac or an EXE file for Windows.

This installation file will work if the machine has Adobe AIR installed. However, if it doesn't have AIR installed, the file won't do anything useful. Fortunately, we can take our installation to yet another level by creating a web-based, seamless install utility called the Adobe AIR *install badge*, which is a convenient way to install your application using a simple, integrated dialog box (called a badge) on a web page. With an install badge, if a user doesn't have Adobe AIR installed, he'll be prompted to download and install it from within the badge installation, making it painless and simple to install your application.

Creating a Seamless Install

To create an installation badge, you need to get the necessary files. Luckily, most of the code needed to prepare the badge has been written for you. You can access these files at the following locations, assuming you've installed Flash Builder at the default location:

Mac

/Mac/Applications/Adobe Flash Builder 3/sdks/3.0.0/samples/badge

Windows

C:\Program Files\Adobe Flash Builder 3\sdk\3.0.0\samples\badge

The badge directories contain several files, but you only need to concern yourself with the following:

AC_RunActiveContent.js

This is a JavaScript file that is used for automatically upgrading Flash Player.

badge.swf

This file contains the necessary code to enable the automatic installation of Adobe AIR if the client doesn't yet have it installed.

default_badge.html

This is a basic HTML file that displays a badge for a seamless AIR installation.

test.jpg

This is a graphic that displays as a representation of the application in *default_badge.html*.

The other files included in this directory are for building your own *badge.swf* file, but typically that isn't necessary and won't be required for this example.

The `default_badge.html` file is a template you can use to create your own install badge on a website. You can test it by loading it in a browser. This HTML file will load the image `test.jpg` and create an Install Now button that a user can click to install the application along with Adobe AIR if it is not installed.

Remember that this set of files is just a template for you to use for your own application. You'll want replace this image with a screenshot of your own application (as 215×100 pixels) so it gives users an idea of what they can expect. You'll need to configure the page to install your own AIR file. Because it's using placeholder values, the `default_badge.html` file has references to a `myapp.air` file that doesn't actually exist. Replace every reference of `myapp.air` with your own application's AIR file, such as `ContactManager.air`. Also, be sure to replace every instance of the words "My Application" with the name of your application, because this name will display in a message under the badge if Adobe AIR is not installed. (By default, the message reads, "In order to run My Application, this installer will also set up Adobe AIR.")

Ultimately, you'll probably want to integrate this sample badge code into your own website, although you can simply provide a link to the `default_badge.html` file.

When you've finished editing the `default_badge.html` file, place these files on your server along with the `ContactManager.air` installation file. When someone loads this HTML file in a browser, the install badge will display as shown in Figure 17-16. If you're curious, we posted our result at <http://learningflex4.com/contactmanager/>. The process for installation is outlined in Figures 17-17 through 17-21.

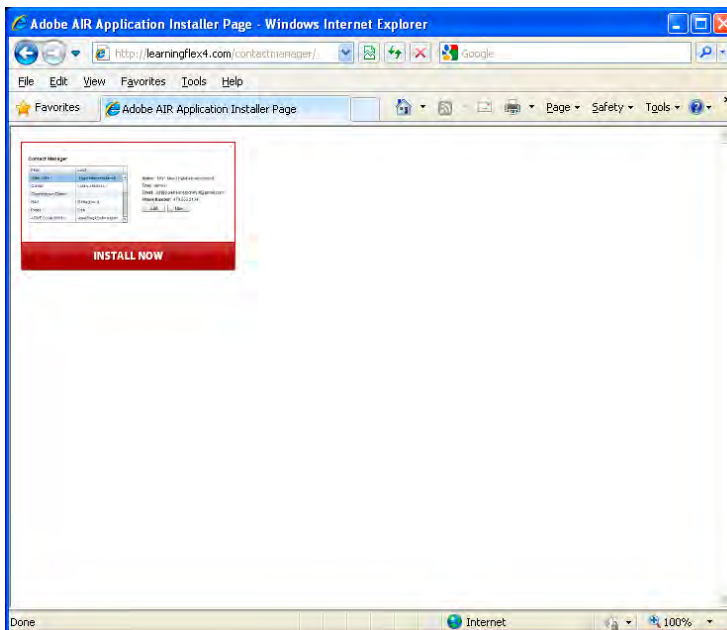


Figure 17-16. The install badge as shown in a web browser

NOTE

If the setup seems correct but you're having trouble with the install badge, try replacing the relative URL (e.g., `ContactManager.air`) with a fully qualified, absolute URL (e.g., <https://www.learningflex4.com/contactmanager/ContactManager.air>). We ran into difficulties combining the badge with a relative URL, and the problem disappeared when we used an absolute URL.

WARNING

In `default_badge.html`, notice the instance of the string `My%20Application`, which is a URL-encoded string. Replace this with your application name—and if you have an application name that includes spaces, it will need to be URL-encoded as well, replacing any spaces with the characters `%20`.

NOTE

This might seem like a lot of trouble to create the installation badge, but remember that this is a cross-platform solution. If you were to create a desktop application using other means, you would—at the very least—be required to create a separate installation file for Mac, Windows, and Linux. Creating an installation badge is a comparatively simple process that lets you create one installation for all platforms.

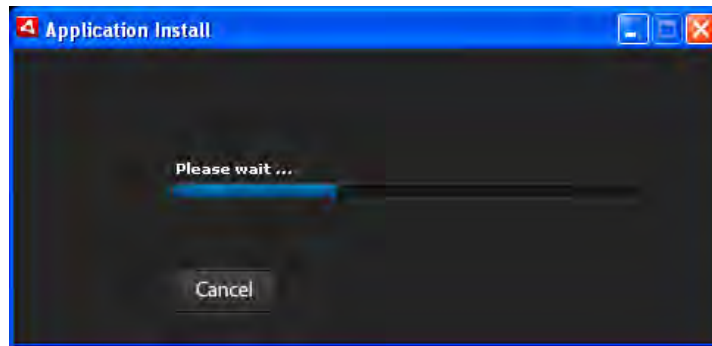


Figure 17-17. Installing an AIR application: Step 1

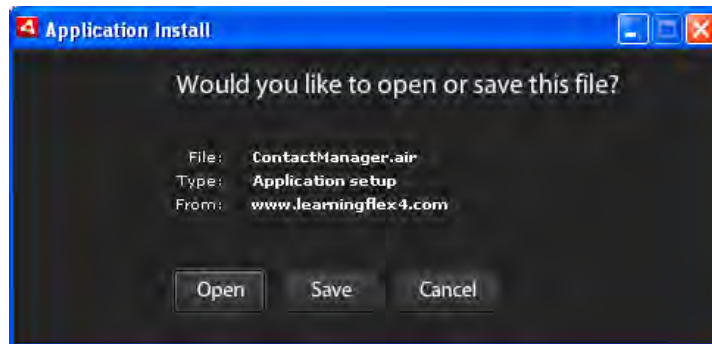


Figure 17-18. Installing an AIR application: Step 2

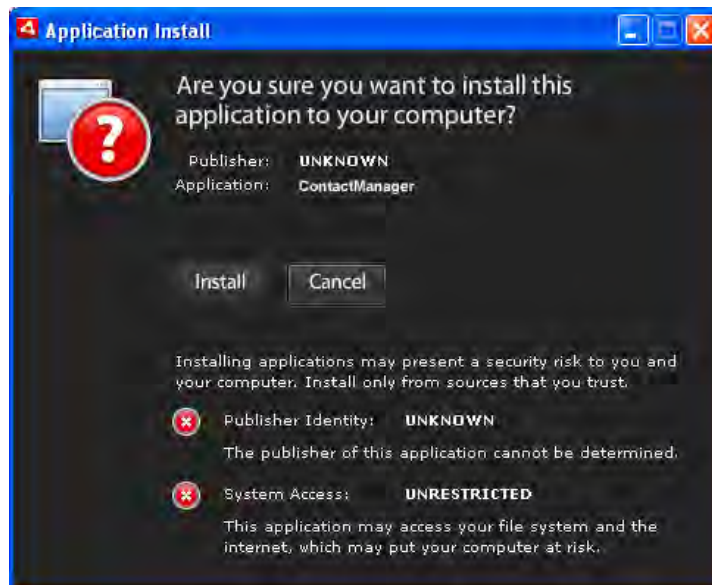


Figure 17-19. Installing an AIR application: Step 3

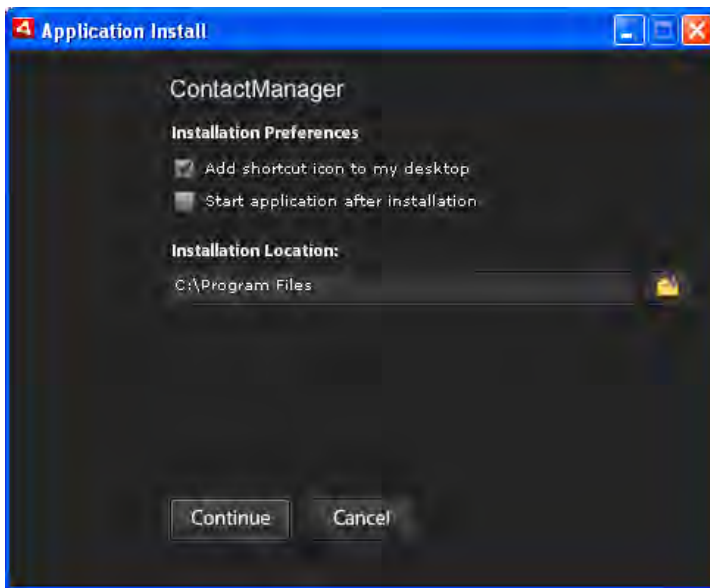


Figure 17-20. Installing an AIR application: Step 4

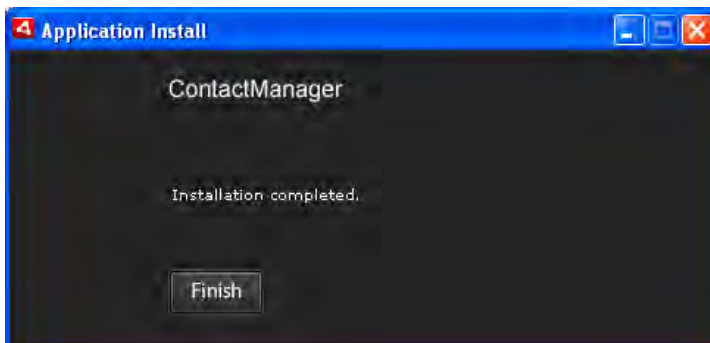


Figure 17-21. Installing an AIR application: Step 5

As we discussed in the previous chapter, Adobe AIR can open up a lot of advanced functionality to your RIAs, including drag-and-drop to and from the desktop, access to the local filesystem and removable media drives, access to the clipboard, etc. The great thing is that everything you've learned about Flex application development transfers perfectly to AIR development.

NOTE

To see samples of Adobe AIR applications, point your browser here: <http://www.adobe.com/devnet/air/?view=samples>.

Now You're Cooking

You've come a long way. Once you start coding your own Flex applications, you may find you need a little help. Need a quick and dirty solution? Try the *Flex Cookbook* online at http://www.adobe.com/go/flex_cookbook.

You don't have to be an expert to get the job done. Here you can search for and grab snippets of code to help solve your problems (at least those dealing with Flex). You'll also learn a lot along the way.

And of course, if you'd like a more tangible reference, grab a copy of the *Flex 4 Cookbook* by Joshua Noble et al. (<http://oreilly.com/catalog/9780596805623/>).

Summary

Congratulations are in order! You've now completed the final step in developing Rich Internet Applications with Adobe Flex 4.

You've gone from getting acquainted with Flex and Flash Builder to learning about MXML, ActionScript, and FXG graphics, and you've acquired the skills necessary to develop complete applications with flexible user interfaces and the ability to connect to data. You've learned to customize your applications using effects, transitions, and stylesheets, and you opened a huge door by hardwiring Flex to a PHP and MySQL server solution, an action that we hope will take you in a new direction. Finally, in this chapter, you gained the skills necessary to deploy your applications on the Web and the desktop.

We hope you enjoyed the book, and we hope to see some of the great applications you'll no doubt create with this powerful—and fun—technology.

In the very last chapter, we dive into a final list of resources and afterthoughts that might guide you as you move along to your next challenge.

WHAT COMES NEXT?

“I had witnessed the start. I was sure of that much.”

—Hunter S. Thomspen

Are you feeling experimental, unstoppable perhaps? If so, that’s great. Those traits certainly describe some of the best developers, and it’s a good indication that you’re ready to tackle a serious project.

On the other hand, maybe you’re feeling a little daunted? If so, don’t worry. Just keep at it. Speaking from our collective experience, the best thing you can do is identify a similar or a related topic and study it, then repeat. Sooner or later something will “click,” and shortly thereafter you’ll experience a fantastic, euphoric moment when an application or key function just works. When it happens, it feels like winning the lottery. So trust us: if this work interests you, it’s worth your while to continue pursuing that magic moment.

If you’re still relatively fresh on the learning curve, remember, the best developers learned to count from 1 before they learned to count from 0. The point is, transitioning from apprentice to master craftsman won’t happen overnight. Like the apprentice, you need time and practice.

So, where do you go from here?

One of the biggest indicators may be decided by your innate preferences. If you hated Chapter 16 (integrating PHP/MySQL), but loved Chapter 14 (effects), then consider finding a collaborator to handle server-side development while you focus on the rich and engaging user experience. Conversely, if you feel comfortable with Flex but find yourself craving more control over data inputs for your projects, furthering your study of PHP and MySQL—or any other server/database combination—would be an excellent decision.

Whatever goals you’re pondering, this chapter provides a cache of information that you might find useful in your quest. We cite some useful APIs and describe them. We reference books that we liked, as well as some suggested by the early reviewers of this manuscript. And finally, we share some online resources you’re likely to appreciate.

IN THIS CHAPTER

Third-Party APIs

Print Resources

Online Resources

Certification

Enfin

Third-Party APIs

With the advent of mature Internet protocols (e.g., SOAP, REST), improved access to broadband connectivity, web-enabled mobile devices, and most recently, affordable mobile broadband, we're finally realizing the true potential of the Internet. This is the Information Age, and appropriately, there's a lot of free information spewing through the Web. All you have to do is capture it for your applications.

The following list of APIs suggests some of the popular services available to you.

AlivePDF (<http://alivepdf.bytearray.org/> and <http://code.google.com/p/alivepdf/>)

Unfortunately, Flex print handling is still fairly underdeveloped, and many developers seem to prefer server-side PDF generation. However, AlivePDF allows for PDF generation through a Flex client application. You can find tutorials on the Web describing how to use this library, but the Flex wizards from Farata Systems—Yakov Fain, Victor Rasputnis, and Anatole Tartakovsky—provide a great demonstration of AlivePDF in their book *Enterprise Development with Flex* (<http://oreilly.com/catalog/9780596154172/>).

as3corelib (<http://github.com/mikechambers/as3corelib>)

This is an ActionScript 3 library with a number of helpful routines, including MD5 and SH1 hashing, image encoders, JSON serializing, and more.

as3flickrlib (<http://code.google.com/p/as3flickrlib/>)

Pull Flickr images into your applications using the Flickr API.

as3syndicationlib (<http://code.google.com/p/as3syndicationlib/>)

If you want to pull Atom RSS feeds into your applications, consider using this library to simplify the heavy lifting. RSS streams can be difficult to parse when they're loaded with hyphens and namespaces, and this library helps you. Oh, and by the way, it's a cousin of as3corelib (see the earlier list item), which you'll need to compile.

ASCB (<http://rightactionscript.com/ascb/>)

This is Joey Lott's library of ActionScript 3.0 classes that support the recipes found in *ActionScript 3.0 Cookbook* (<http://oreilly.com/catalog/9780596526955/>).

asSQL (<http://code.google.com/p/assql/>)

This library, which allows direct connections between Flex/ActionScript and MySQL, PostgreSQL, and Microsoft SQL Server, has a lot of potential, especially if combined with a cross-domain policy file on a distributed server. As their website states, ServerBolt (<http://www.serverbolt.com/>)

services/custom-hosting) offers web hosting options that include cross-domain policy support.

Astra Web API (<http://developer.yahoo.com/flash/astra-webapis/>)

We experimented with this Yahoo! API when we built the **YahooSearch** application (Chapter 11), but it also supports Yahoo! Answers, Weather, and Calendar services.

Facebook ActionScript API (<http://code.google.com/p/facebook-actionscript-api/>)

If you want to build widgets for Facebook, start here.

flexlib (<http://code.google.com/p/flexlib/>)

A community-driven code library offering a number of specialized custom Flex components. If you're feeling up to the challenge, they consider submissions!

Google Maps API (<http://code.google.com/apis/maps/documentation/flash/>)

Certainly a mashup favorite, the Google Maps API provides a spatially aware map portal and a set of essential geographic functions you can use for a myriad of possibilities. One word of caution, though: their licensing terms require your map to be free to use and public-facing. If you intend to embed Google Map functionality within an enterprise application, you're required to purchase a license that allows commercial uses.

tweener (<http://code.google.com/p/tweener/>)

Tweener is a custom ActionScript library with classes designed to simplify complex animations.

Twitter ActionScript/Flash APIs (<http://dev.twitter.com/pages/libraries#flash>)

You have a few options when it comes to connecting to the increasingly popular Twitter service. Visit their site for details.

UMap API (<http://www.umapper.com/pages/developers/>)

This API provides wholesale access to many mapping services, including OpenStreetMap, Bing Maps, MapQuest, and others. If you have commercial interests in mind, this API is worth a look because the pricing structure is *per request*, meaning if you're not submitting a tremendous number of searches (e.g., dragging the map), you pay less.

Yahoo! Maps API (<http://developer.yahoo.com/flash/maps/>)

Like its Google counterpart, the Yahoo! Maps API provides an advanced online mapping utility full of general-purpose functions you can use to create compelling mashups and public-facing Location-Based Services (LBS). The service is free for personal use, but again, be careful and be wise about using it for commercial purposes.

YouTube API (<http://code.google.com/p/as3youtubelib/>)

Use this API to pull YouTube videos into your Flex applications.

Print Resources

The Stacks Effect:

A learning phenomenon describing accelerated discovery related to library-based research. Also widely believed to occur between the Computing/IT shelves at your local book retailer.

We love books. Some programmers will argue that the Internet is the best way to research programming and software development. They would be correct in claiming that the Internet provides a handsome wealth of information about programming, but at the very best, it's a 49/51 trade-off compared to print media. (So, maybe we're imposing a little bias here!)

Sure, you can use Ctrl-F to search a web page for keywords. Sure, you can jump through a 1,000-page PDF in the time it takes to type the search filter. But what about when you're not yet knowledgeable enough to know which topics to search? We'll tell you what: you get nowhere fast and waste your time doing it. Also, with web-based learning, you're only exposed to the details of your search, so a breadth of knowledge can be difficult to acquire.

With a book, however, you learn a lot just by flipping through the pages. Books provide an organized approach to learning, one that is conducive to broad discovery. Books pick an audience and then cater to the needs of that audience from start to finish. You can grab a pen and add your own thoughts to books. You can fold down a page, and later, when you want to remember the contents of that page, you can open the book to find that topic quicker than you can launch an Internet browser and call up a bookmark. You can read a book while the Internet is down. Books can't get viruses. Books are tangible; they don't disappear when domains expire, and they can't be overwritten or renamed accidentally. Unless explicitly destroyed, books last forever. We love books.

With this point so deftly imposed on our readership—if not merely for the sake of ending some arguments at work—here is a list of books we appreciate:

Tariq Ahmed et al., *Flex 4 in Action* (Manning Publications, 2010).

Joshua Noble et al., *Flex 4 Cookbook* (O'Reilly Media, 2010).

David Gassner, *Flash Builder 4 and Flex 4 Bible* (Wiley, 2010).

Mike Jones, *Developing Flex 4 Components* (Addison-Wesley, 2010).

Darren Richardson et al., *Foundation ActionScript 3.0* (friendsofED/Apress, 2009).

Colin Moock, *Essential ActionScript 3.0* (O'Reilly Media, 2007).

William Sanders and Chandima Cumararatunge, *ActionScript 3.0 Design Patterns* (O'Reilly Media, 2007).

Joey Lott et al., *ActionScript 3.0 Cookbook* (O'Reilly Media, 2006).

Matthew Keefe, *Flash and PHP Bible* (Wiley, 2008).

Jono Bacon, *Practical PHP and MySQL* (Prentice Hall, 2006).

Larry Ullman, *PHP & MySQL* (Peachpit Press, 2003).

Ben Forta, *MySQL Crash Course* (Sams, 2005).

Of course, there are many other great books on these subjects, but the list provided here includes those works that the authors of this book and our early reviewers hold in high regard. We surely welcome you to swing by this book's WordPress blog (<http://learningflex4.wordpress.com/>) and put in a good word for the favorites on your reading list.

Online Resources

Contrasting the argument we posited in the previous section, it remains true that user-driven content provided by discussion boards, blogs, and the wiki model altogether creates a fantastic lexicon of documentation. In other words, there are a lot of bloggers and forums covering Flex/ActionScript development that you might want to follow. Also, Adobe has provided a significant body of official Flex documentation that you should collect, much of which is available through Internet websites or as PDF downloads.

Articles and Blogs

The following articles and blogs do a fantastic job of imparting knowledge, and many of the following posts are targeted at topics we missed; as such, we recommend that you give them a look.

“Flex 4 and the Text Layout Framework,” Holly Schinsky

http://www.adobe.com/devnet/flex/articles/flex4_textlayout_framework.html

“Getting at your SQL Database using Flex and .NET Web Services,” Michael Fitchett

<http://www.fitchett.me/index.php/development/flex-development/getting-at-your-sql-database-using-flex-and-net-web-services/>

“Consume .NET-based web services written in C#,” Nishad Musthafa

http://www.adobe.com/devnet/flex/articles/flashbuilder_webservice_dotnet.html

“PHP as a data source for Flex applications,” Robert Bak

<http://insideria.com/2010/06/a-whole-lot-of-people.html>

“Flex DataGrid Goodies—Row Color and Others (Hacking a DataGrid),” Charlie Key

<http://www.switchonthecode.com/tutorials/flex-datagrid-goodies-row-color-and-others>

“28 Rich Data Visualization Tools,” Theresa Neil*<http://insideria.com/2009/12/28-rich-data-visualization-too.html>****“Effects in Adobe Flex 4,” Chet Haase***

Part 1: Basic effects

http://www.adobe.com/devnet/flex/articles/flex4_effects_pt1.html

Part 2: Advanced graphical effects

*http://www.adobe.com/devnet/flex/articles/flex4_effects_pt2.html****Flex Examples, Peter DeHaan****<http://blog.flexexamples.com/>****Flex-Blog.com, various authors****<http://www.flex-blog.com/>***Reference Material**

Adobe manages a ton of online reference material documenting Flex, ActionScript, and AIR. You could assemble a bookmark collection like this without too much difficulty, but sometimes it's nice to see everything in one place on the printed page.

We've ordered these links so that those we believe will benefit you the most appear first.

Flex 4 Language Reference*<http://livedocs.adobe.com/flex/gumbo/langref/>****ActionScript3 Language and Components Reference****<http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/>****Using Adobe Flex 4 (2,444 pages of free material!)****http://help.adobe.com/en_US/flex/using/flex_4_help.pdf****ActionScript 3.0 Developer's Guide (1,074 pages of free material!)****http://help.adobe.com/en_US/as3/dev/as3_devguide.pdf****Accessing Data with Adobe Flex 4 (133 pages)****http://help.adobe.com/en_US/Flex/4.0/AccessingData/flex_4_accessingdata.pdf****Using Adobe Flex 4 Compiler API (27 pages)****http://help.adobe.com/en_US/Flex/4.0/CompilerAPI/flex_4.0_compilerapi.pdf****Adobe Flex Coding Conventions****<http://opensource.adobe.com/wiki/display/flexsdk/Coding+Conventions>*

Flex.org Resources Page<http://flex.org/resources>**Flex Developer Center**<http://www.adobe.com/devnet/flex/>**AIR Developer Center**<http://www.adobe.com/devnet/air/>**Adobe Flex Cookbook**<http://cookbooks.adobe.com/flex>**Adobe AIR Cookbook**<http://cookbooks.adobe.com/air>**All documentation relevant to Flex 4, zipped (59.4 MB)**http://www.adobe.com/go/learn_flex4_alldocumentation_en**Plotter-size poster of the Flex 4 framework**http://www.adobe.com/devnet/flex/pdfs/flex4_poster_web.pdf**Plotter-size poster of ActionScript classes**http://www.adobe.com/devnet/flex/pdfs/actionscript_poster_web.pdf

Certification

If you love programming in Flex but are having a difficult time getting your foot in the door somewhere, consider a personal study aimed toward Adobe Certified Expert (ACE) certification. Even if you aren't interested in taking the test, preparing for the exam by studying its content areas provides a well-structured approach to mastering Flex.

The test is composed of 59 multiple-choice questions, and it requires you to select a correct line of code among some clever imposters. With a \$150 fee to enroll, the test isn't a joke. Table 18-1 presents the overall exam structure, but you'll find a better breakdown by topic if you download the study guide from the link provided after the table.

Table 18-1. *Adobe Certified Expert (ACE) exam structure*

Topic area	% of exam	# of questions
Creating a user interface (UI)	34	20
Flex system architecture and design	17	10
Programming Flex applications with ActionScript	20	12
Interacting with data sources and servers	14	8

Overview

http://www.adobe.com/devnet/flex/articles/flex_certification.html

Study guide

http://www.adobe.com/devnet/flex/pdfs/ace_exam_guide_flex4.pdf

Signup (Pearson VUE)

<http://www.pearsonvue.com/adobe/>

Enfin

“All programmers are optimists. Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundreds of nitty frustrations drive away all but those who habitually focus on the end goal.”

—Frederick Brooks

It’s a curious thing to write a book. In our closing thoughts, it’s impossible not to wonder if we achieved our vision—helping the new developer get up to speed. We hope you’ve enjoyed our journey, and hope it’s benefited you. If you’re inclined, we welcome hearing from you via the companion website (<http://www.learningflex4.com/>) and forum (<http://learningflex4.wordpress.com/>).

As you turn the final page, we congratulate you on your accomplishment and, as it goes with everything, *bon chance, toujours*.

Blue skies, even.

—Alaric Cole and Elijah Robison, November 2010

CREATING A DEVELOPMENT ENVIRONMENT

The *production environment* refers to the specific hardware, software, network configuration, and distributed system architecture affecting any host system(s) that will serve a RIA. With that said, a *development environment* is simply the best *imitation* of the production environment a developer can piece together on her local system or network. The perfect development environment exactly replicates hardware specs, choice of operating systems, and software versions composing the production environment.

As you can imagine, a number of factors make the perfect development environment unattainable. The usual suspects are budget constraints and the impracticalities of maintaining a separate yet specialized system to support testing scenarios. So, what constitutes a good compromise? Well, try to match the server software as much as possible, preferably getting down the software versions.

Since we're using PHP and MySQL for our host applications, there is a quick and easy shortcut we can take to weave a development environment right into our local machine. If you run Windows, it's called WAMP, and if you run Mac OS, it's called MAMP.

Use WAMP (Windows) or MAMP (Mac OS)

Hopefully we didn't worry you with our discussion of development servers. Rigid development environments are usually the domain of corporate programming teams and enterprise application developers. The truth is, most of us are comfortable developing against a more casual test bed—we're talking about WAMP or MAMP.

The important part of the acronym is "AMP," which stands for Apache, MySQL, and PHP, three free server technologies that form the backbone of the Internet. Both WAMP and MAMP provide "canned," easy-to-download, ready-set-go installers that weave an Apache, MySQL, and PHP bundle directly onto your local machine. Depending on your choice of OS, here are the URLs for WAMP and MAMP, respectively:

IN THIS CHAPTER

Use WAMP (Windows) or MAMP (Mac OS)

Add PHP Development Tools (PDT) to a Flash Builder Installation

Summary

NOTE

Your development environment priority: match development server software to the versions running on the production server. If you can't get a precise version match, get as close as possible.

NOTE

If you have an old system collecting dust in the closet, you can make a fun weekend project out of converting it into a Linux server, complete with individual PHP and MySQL installations, and then using it as a standalone development server.

WAMP (Windows)

<http://www.wampserver.com/en/index.php>

MAMP (Mac OS)

<http://www.mamp.info/en/index.html>

Your first steps should be downloading and installing the appropriate software bundle. If you need assistance, either site provides reasonable documentation, including installation instructions:

WAMP documentation

<http://www.wampserver.com/en/presentation.php>

MAMP documentation

<http://documentation.mamp.info/en/mamp>

Our discussion of WAMP only considers a Windows installation, and we're assuming a default installation at that. So, if you desire to use MAMP on Mac OS, be sure to visit the MAMP web page and follow their installation instructions.

Inspecting the WAMP Tool Menu



Figure A-1. The WAMP taskbar icon

When you run WAMP on Windows, it'll add a taskbar icon (Figure A-1) that gives you quick access to the server functions. Left-clicking the icon will open a menu containing options (Figure A-2).

Let's break down the options available to you in the WAMP menu:

Localhost

In the WAMP menu, Localhost calls the browser to open the default page in the web root directory. When you install WAMP, it adds *index.php* to the web folder, which opens the page depicted in Figure A-3.

phpMyAdmin

This option launches phpMyAdmin, which is a PHP-based portal into your local MySQL installation. In short, it provides GUI tools you can use to create databases and tables, manage users, and populate tables with records (Figure A-4).

www directory

Opening the *www* directory takes you directly to the web root, which is any directory in the local filesystem recognized by WAMP for server processing and HTTP request handling (Figure A-5). Any files in or below the web root directory are said to be in the *web scope*.

Quick Admin

The Quick Admin functions give you easy access to the start/stop actions for the various server utilities.



Figure A-2. The WAMP functions menu

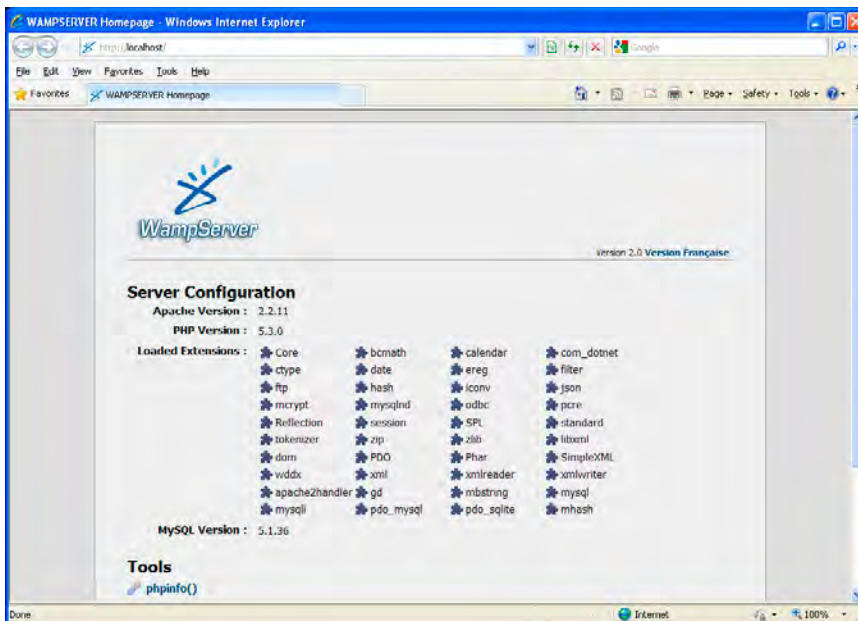


Figure A-3. The default page provided with a fresh WAMP install

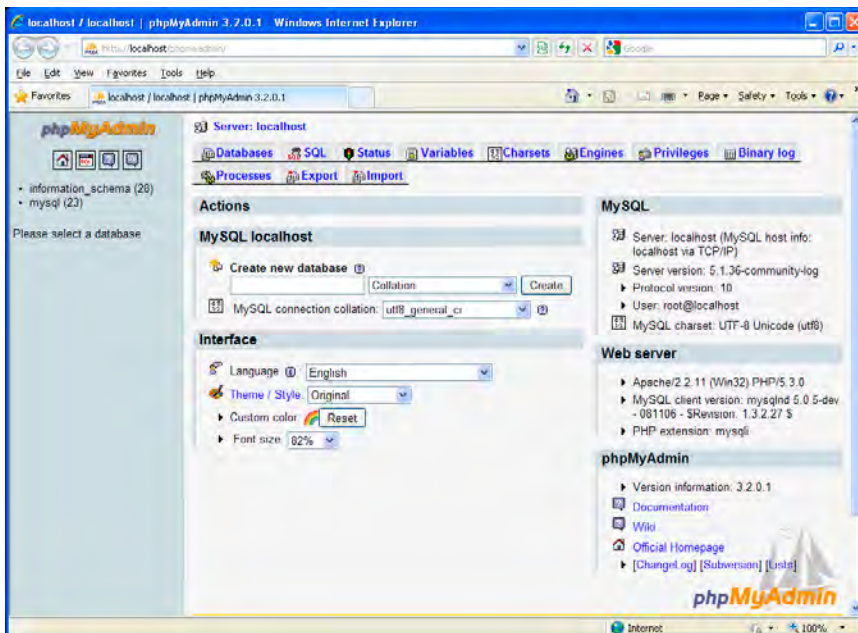


Figure A-4. Accessing the MySQL installation using phpMyAdmin

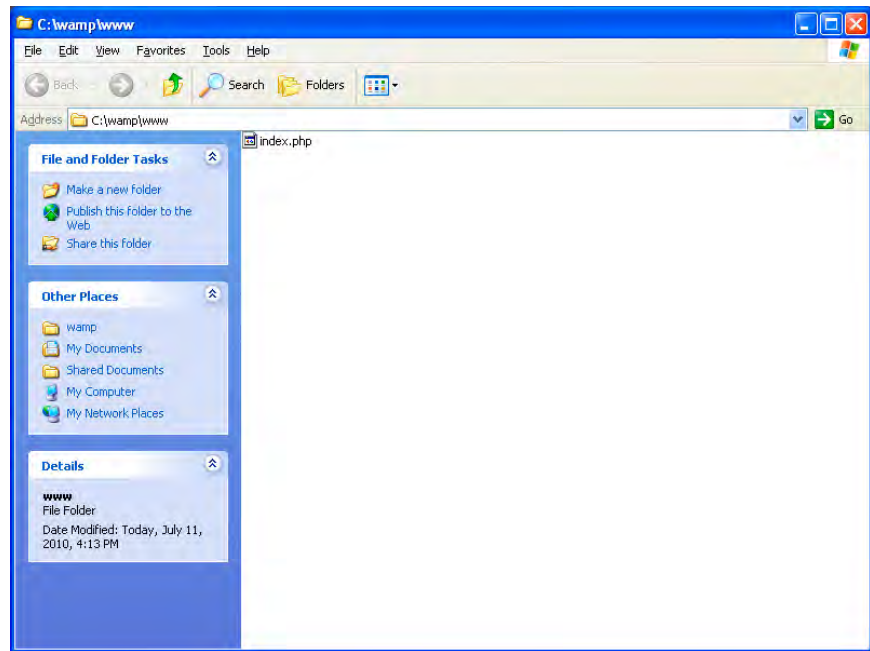


Figure A-5. A brand-new web root folder with default contents

WARNING

If WAMP fails to open the default start page following a fresh install, try stopping and restarting Apache and PHP. If that doesn't fix it, check for conflicts on port 80. If you're running Skype, we've noticed it has a tendency to fight WAMP, so try exiting Skype and restarting WAMP. Other likely culprits include firewalls, anti-virus software, peer-to-peer clients, and any other thing that might attempt to borrow port 80 for its own benefit.

Test the Installation

Now that you've installed WAMP and inspected the menu options, go ahead and test your installation. To start, open the WAMP menu and click on Localhost. This should open *index.php*, which WAMP included with your installation (Figure A-3).

If the WampServer page loaded, your development environment is ready for use.

Add PHP Development Tools (PDT) to a Flash Builder Installation

In this section we install the PHP Development Tools (PDT) plug-in for Eclipse into your Flash Builder installation. In addition to a few extra menus and functions, PDT extends Flash Builder to include PHP syntax coloration, and as a bonus, PDT also installs XML Developer Tools for Eclipse, which add not only XML syntax coloration, but also unique Source and Design views that you may find helpful when working with XML files.

Installing PDT

Assuming you have Flash Builder running, open the Help menu and select Install New Software (Figure A-6). In the Install dialog, click on the Add button (Figure A-7). When the Add Site dialog opens, add the Galileo Update Site as you see in Figure A-8.

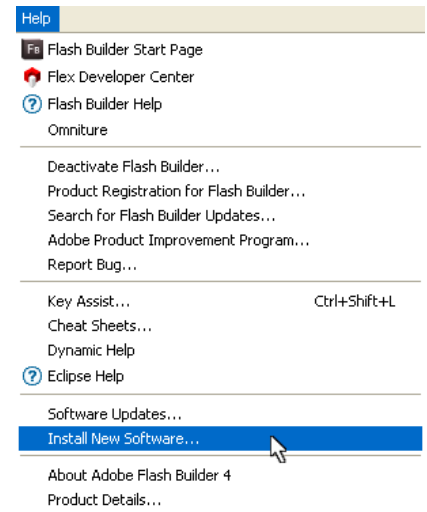


Figure A-6. Accessing the Install New Software feature

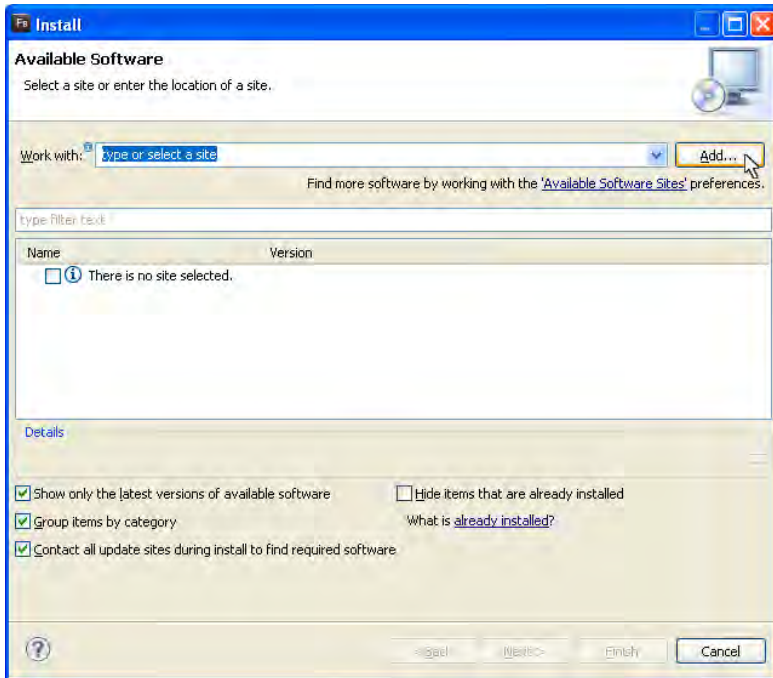


Figure A-7. Preparing to add a new installation source path

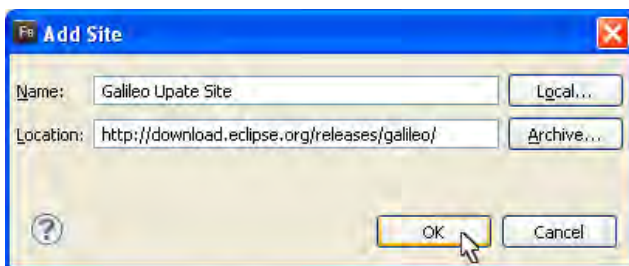


Figure A-8. Adding the Galileo update site and source location

Notice these settings in Figure A-8:

Name: Galileo Update Site (this can be anything you want)

Location: <http://download.eclipse.org/releases/galileo/>

When you return to the Install dialog, Flash Builder might be in the middle of processing responses from the Galileo update site. Even if Flash Builder is processing, change the “Work with” option to “--All Available Sites--”, and then type **pdt** into the search field, as depicted in Figure A-9.

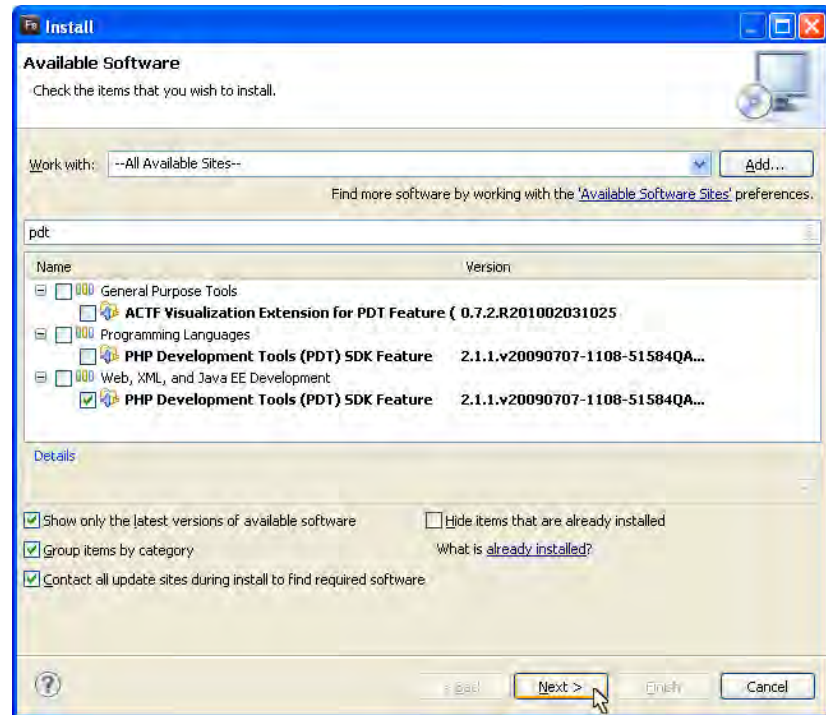


Figure A-9. Searching available plug-ins against “pdt” and selecting an instance of PHP Development Tools (PDT) from the options

You should receive a number of options, even after filtering the available software against the keyword “pdt”. You’ll likely have a valid PDT option under multiple categories. It doesn’t matter which PDT entry you check, just pick one and click Next, as shown in Figure A-9.

In the next screen, you’ll be presented with a review of what you’re installing (Figure A-10). Note that Eclipse XML Editors and Tools is included, along with a handful of other goodies. Just click on Finish to complete the process.

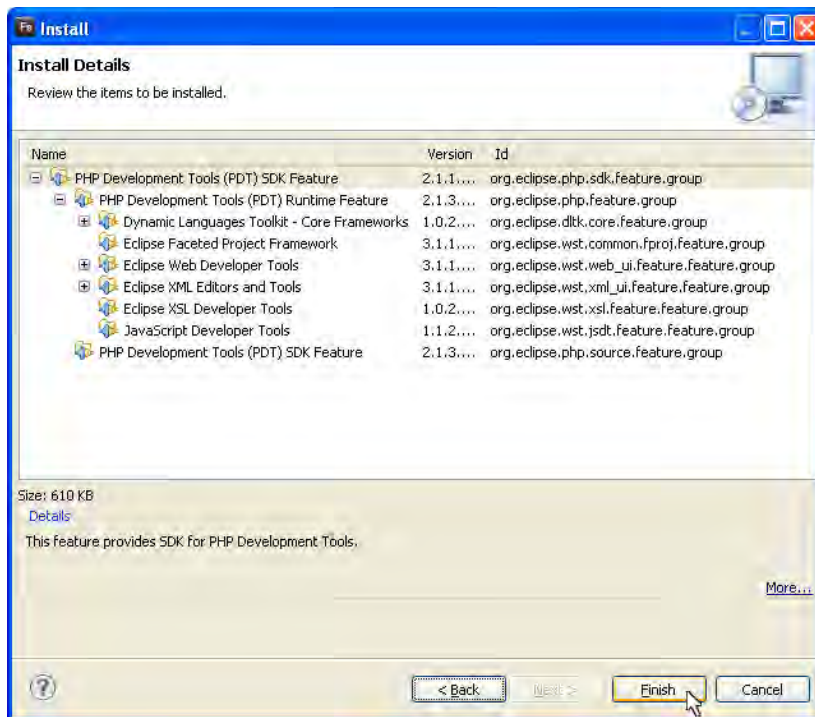


Figure A-10. Confirming the PDT installation and reviewing additional utilities included with the package

Testing the XML Developer Tools

Before we experiment with something unfamiliar, let's gain some confidence by looking at something we're familiar with—specifically, XML.

To get a feel for the features you gain from the Eclipse XML Developer Tools, return to the **PhotoGallery** application you developed in Chapters 12 and 15. Once the project is available, open *photos.xml*. Depending on whether Flash Builder opens the XML file into the new Design view or Source view with syntax coloration, you might need to switch views (Figure A-11).

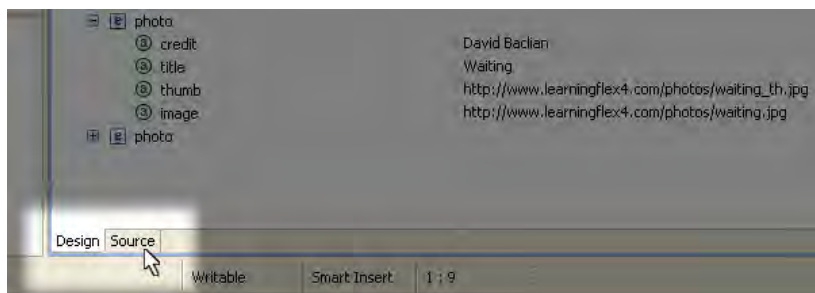


Figure A-11. Switching between Design and Source views with the XML Developer Tools

In Design view, notice that you have a fresh new graphical interface to use when writing XML. In Figure A-12, we're adding a new **photo** node via the context menu (in Windows, right-click; in Mac OS, Control-click). You can also update the attribute and content values in this view as if you're working with input fields.

Not everyone will be enthused about using a GUI menu to modify XML, though, and for those people, there's also an XML Source view. The new XML Source view improves upon Flash Builder's default rendering of XML. To start, XML Source view provides syntax coloration, which is always a good thing (Figure A-13). You'll also get code completion assistance once you establish a pattern of nodes and attributes.

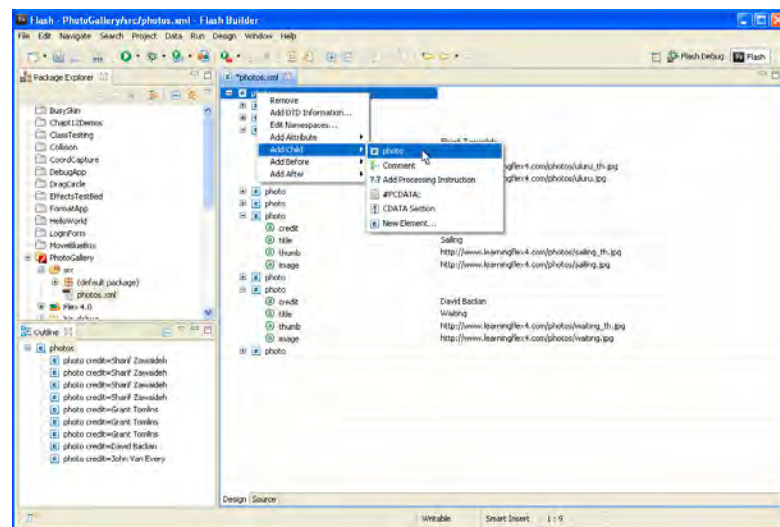


Figure A-12. Adding a new photo node using the GUI tools in XML Design view

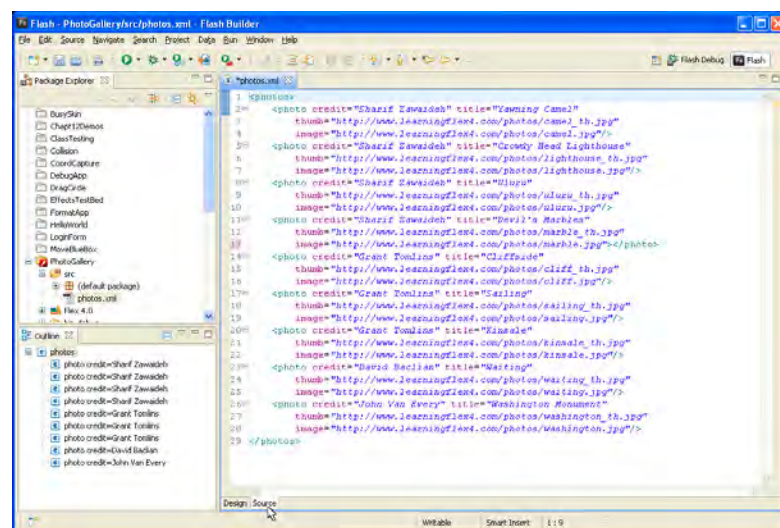


Figure A-13. Appreciating the XML syntax coloration in XML Source view

Testing the PHP Development Tools

Since we're testing things, let's create a quick Flex project called **HelloPHP** to see the PHP syntax coloration and to tie WampServer into something.

Start by creating a new Flex project. This time, however, note the “Server technology” option toward the bottom of the dialog. As shown in Figure A-14, change the “Server technology” option from None/Other to PHP.

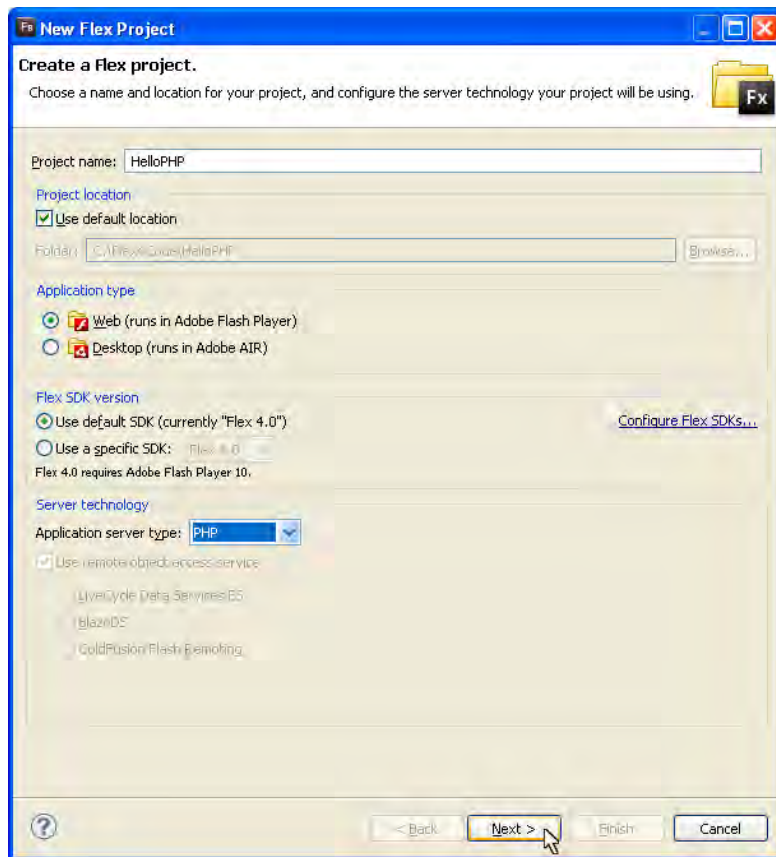


Figure A-14. Creating a new Flex project to use PHP server technology

In the next screen you'll be prompted to validate your local development server (Figure A-15). Assuming a default WAMP installation (adjust accordingly for MAMP, of course), configure the “Web root” option to `C:\wamp\www` (i.e., the root of the web scope), and set the Root URL to either `http://localhost` or `http://127.0.0.1`, which are synonymous URLs pointing to the domain root of your local server environment. The “Output folder” value should be added by Flash Builder; you can modify it if you want, but we'll go with the default in our example.

NOTE

Need help remembering the localhost IP equals 127.0.0.1? Check out the following link; it's about as cool as \$2.99 can be, and as far as laptops are concerned, it's one size fits all (Netbooks need not apply): <http://www.thinkgeek.com/homeoffice/stickers/5fa6/>.

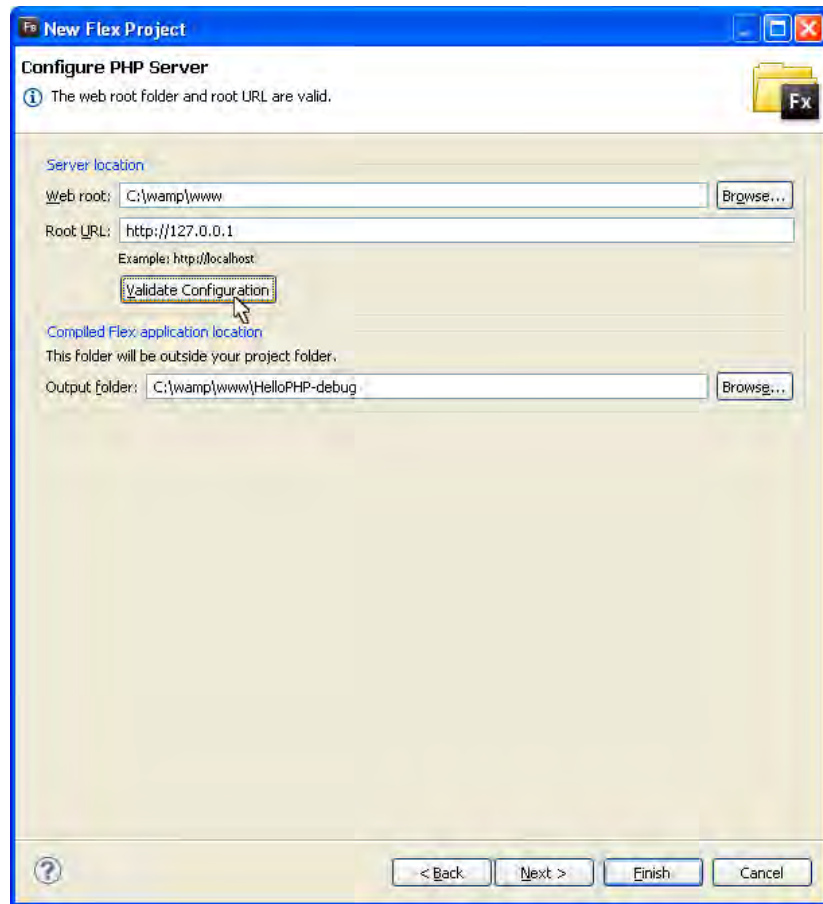


Figure A-15. Configuring and validating the local development environment

Once you finish entering values, click **Validate Configuration** so Flash Builder can test your server. Make sure WAMP is installed, running, and “online”; otherwise, Flash Builder won’t be able to confirm your server setup. If you pass the test, click **Finish** to complete the **HelloPHP** project creation.

Before we worry about any Flex code, let’s create an ultra-simple PHP script to return the phrase “Hello from PHP!” to whatever calls it.

In the Package Explorer pane, select the project’s *src* directory, open the File menu, and then select **New**→**Folder**. Call the folder *php*. You should now have a new folder called *php*. Select the *php* folder, then follow **File**→**New**→**Other**, and within the *php* folder, select **PHP File** and click on **Next** (Figure A-16).

In the next dialog, **New PHP File**, note that the Source folder is already provided; simply replace *newfile.php* with *hello.php*, preserving the *.php* extension as shown in Figure A-17, and then select **Finish**.

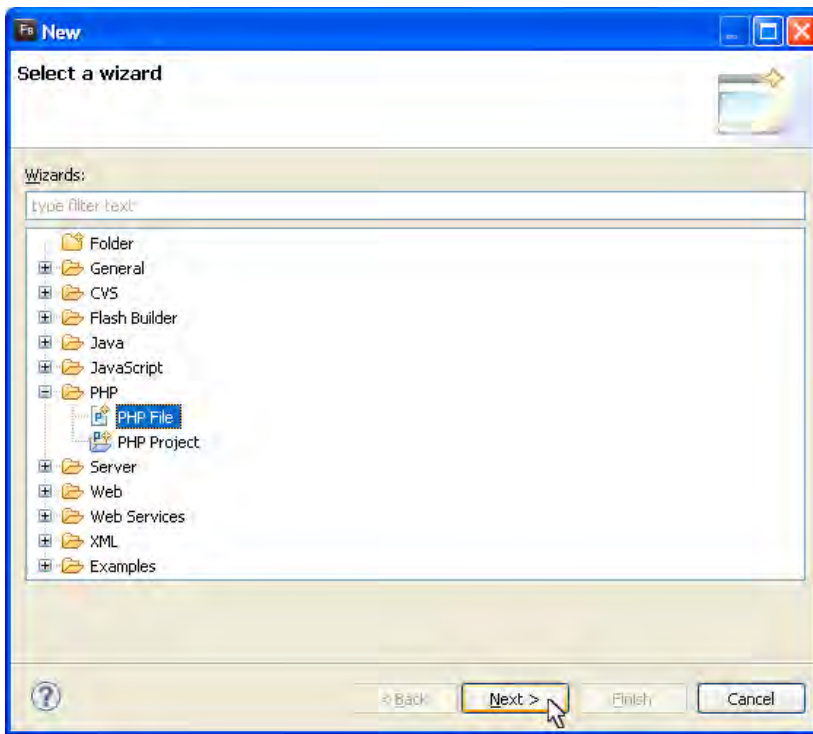


Figure A-16. Creating a new PHP file with the help of PDT menus

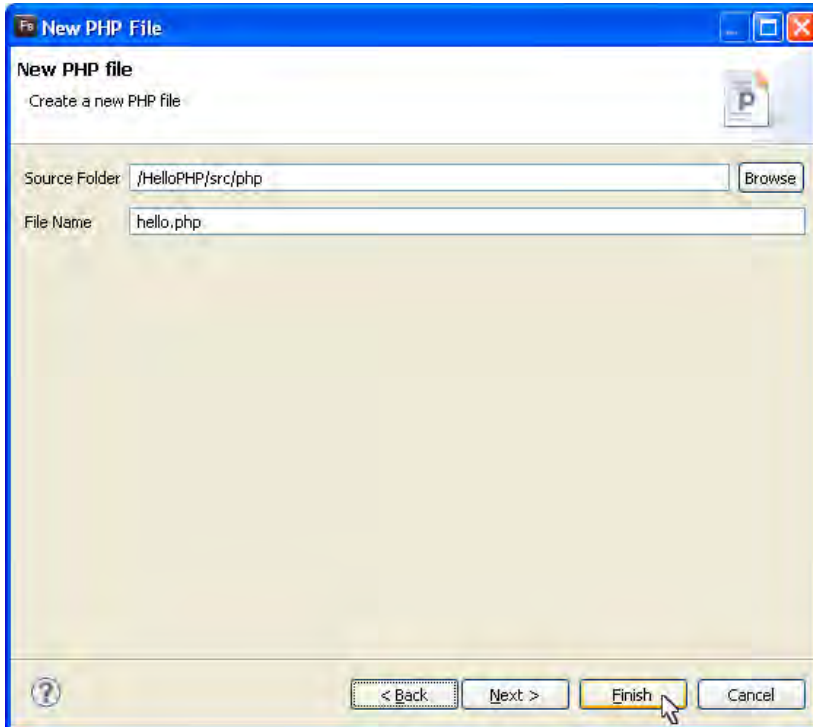


Figure A-17. Naming the new PHP file and adding it to the project

When you emerge into an Editor panel, add the code from Example A-1 to complete *hello.php* (Figure A-18).

Example A-1. *HelloPHP\src\php\hello.php*

```
<?php
    echo "Hello from PHP!";
?>
```

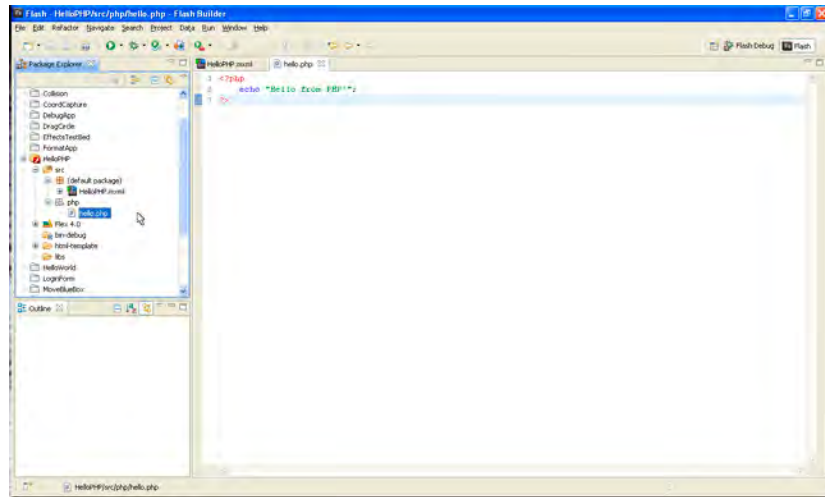


Figure A-18. A very simple PHP script with syntax coloration

Finally, to complete the **HelloPHP** application, replicate the code in Example A-2 into the project's application file.

Example A-2. *HelloPHP\src\HelloPHP.mxml*

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <fx:Script>
        <![CDATA[
            private function onEcho():void{
                resultTI.text = phpService.lastResult.toString();
            }
        ]]>
    </fx:Script>

    <fx:Declarations>
        <s:HTTPService id="phpService" url="php/hello.php" result="onEcho()"/>
    </fx:Declarations>
```

```

<s:layout>
  <s:VerticalLayout horizontalAlign="center" verticalAlign="middle"/>
</s:layout>

<s:TextInput id="resultTI"/>

<s:Button label="Call PHP Script" click="phpService.send()"/>

</s:Application>

```

By now, we hope you're familiar enough with Flex that this code is not too mysterious.

Essentially, we're using an **HTTPService** component to call the PHP file we just created. Within the runtime, we'll use a **Button** to call the service component's **send()** method, for which we arranged the service component's **result** event to call the **onEcho()** function from the application's Script/CDATA block. In turn, the **onEcho()** function assigns the **TextInput** control's text property the service component's **lastResult** value. Note that we are converting the **lastResult** to a **String** using the **toString()** method of the **Object** type. Figure A-19 shows the result.

Summary

In this appendix, we helped you become familiar with using WAMP or MAMP for your local development environment. We also installed the PHP Development Tools (PDT) plug-in for Eclipse and tested it within a Flash Builder installation. In the process, you learned about the extra features PDT adds to Flash Builder/Eclipse that assist with both XML and PHP file creation and editing. Additionally, we created the **HelloPHP** Flex application to ensure the WAMP environment was running and to briefly demonstrate calling PHP scripts from a Flex application using the **HTTPService** component.

For those so inclined, Appendix B provides a short primer on using phpMyAdmin to access and manage a MySQL database right from within the WAMP environment.

NOTE

It's worth emphasizing this point: we can call the PHP file using a relative URL, `php/hello.php`, only because Flash Builder helpfully copies and compiles our project to an external Output folder that's within WampServer's web scope.

If you try testing a compiled application in your local filesystem against a PHP file on a remote web host, you would need to use a fully qualified URL, such as `http://www.learningflex4.com/php/hello.php`.

By the way, we made `hello.php` available from that location, so you are welcome to pull it into your demo application for testing.

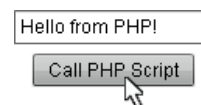


Figure A-19. Calling `hello.php` from a Flex application at the click of a button

MYSQL BASICS

The first half of this appendix introduces you to three essential SQL operations—**SELECT**, **INSERT**, and **UPDATE**—and the second half of the appendix demonstrates how to use phpMyAdmin to create and manage a MySQL database within the local WAMP/MAMP environment. As usual, let's have a quick discussion before we get into the dirty work.

A solid database administrator (DBA) can completely manage his database using a command-line interface. Having said that, anyone serious about learning SQL should probably start at the command line and switch to a GUI application only once he's comfortable reading and hacking SQL in a terminal window.

Here's the thinking: if you can handle SQL using the command line, you're more likely to succeed later when you write SQL code inside functions of other languages such as PHP, which use your scripted SQL to communicate with the database. In a perfect world you've already covered the first few chapters of Ben Forta's *MySQL Crash Course* (Sams) before ever acquiring our book. Back in reality, however, we realize this might be your first look at some SQL.

If you don't intend to be a true database gangster, and if you only intend to call SQL statements from your own applications, here's our position in two statements:

- You *don't* need to know how to create databases, tables, table fields, new users, and user privileges with script.
- You *do* need to know how to write queries in script.

Having said that, the context of our discussion will focus exclusively on using the **SELECT**, **INSERT**, and **UPDATE** operations, which we will ultimately call from PHP in Chapters 16 and 17's **ContactManager** application. To get started, we discuss language elements and syntax rules that apply to MySQL scripts.

IN THIS CHAPTER

Language Elements and
Syntax

MySQL Statements

Creating a Database with
phpMyAdmin

Summary

NOTE

GUI MySQL applications include the favorite, MySQL Administrator, as well as MySQL Workbench, Navicat, and phpMyAdmin. Because many web hosting packages make phpMyAdmin available as an administrative tool, and further because it's prepackaged with WAMP and MAMP, a little later we demonstrate how to use phpMyAdmin to create and modify a MySQL database.

NOTE

At this point, Ben Forta's MySQL Crash Course (*Sams*) might not cover the latest MySQL release, but it does such a fantastic job of teaching MySQL to new database users that it still merits our recommendation. The SQL basics don't change between revisions, only the deeper aspects. So if you find a used copy of MySQL Crash Course for a good price, grab it as a solid getting-started guide. Just remember, as you get deeper into SQL, you may benefit from a newer book; otherwise, you could run into some confusing situations.

SQL, DML, and DDL

Don't be surprised if you catch developers referring to **SELECT**, **INSERT**, **UPDATE**, and **DELETE** statements altogether as "queries" while casually discussing SQL. Oftentimes DBAs will refer to any statement sent from an application to a database as a "query." Case in point: the PHP function `mysql_query()` (which we discuss in Appendix C) can submit any of these statement types. However, formally speaking, only the **SELECT** statement constitutes a true query. Meanwhile, **INSERT**, **UPDATE**, and **DELETE** are considered *Data Manipulation Language* (DML), which is a subcategory of SQL and includes any database activity that results in changes to data records.

Yet another subcategory of SQL, called *Data Definition Language* (DDL), refers to database statements that handle schema or user management. To help understand the word *schema* in the context of a database, imagine the schematic of an electrical circuit; it's a diagram showing every path between the circuit's component parts. A database schema is similar, as it refers to the unique layout of tables, their field names and data types, and the various linkages between them (called joins). Keywords for this statement type include **CREATE**, **DROP**, and **ALTER**.

Language Elements and Syntax

SQL scripts/statements are keyword-driven, and like in sentence structure, the typical query contains specialized "chunks" or *clauses* that have a unique purpose in the statement. Here's a simple selection statement:

```
SELECT * FROM contact ;
```

The statement translates to:

"SELECT * (all fields) FROM (the table) contact ; (end statement)"

We can extend the selection statement to include a *predicate*, which generally introduces a condition or limitation, but can also perform organizing operations. In this case, we'll use a **WHERE** clause to impose a predicate expression.

```
SELECT * FROM contact WHERE first LIKE 'jo%' ;
```

The **WHERE** clause translates as:

"... WHERE (the value in column) first (is) LIKE 'jo%' ;"

The percent (%) symbol is a *wildcard* operator, and with the wildcards in this context, the condition targets any record where the contact's first name, which is stored in the field named "first", has the letter combination "jo". Since a wildcard appears before and after the letter combination, the statement looks for "jo" to occur anywhere in the first name. So, John, Billy Joe, Leah-Jo, and Major would all qualify.

NOTE

In SQL terms, "column" and "field" are synonymous, as are "row," "record," and "tuple."

Here are a few more details you need to be aware of regarding MySQL statements:

MySQL is not case-sensitive

Although we capitalized the keywords, you could create every statement using only lowercase. However, convention dictates capitalization of your SQL keywords.

Whitespace and line breaks are ignored

That means you can use whitespace and line breaks to format an SQL statement for improved readability. As such, the statement we explored previously could be rewritten as:

```
SELECT *  
FROM contact  
WHERE first LIKE '%jo%' ;
```

The semicolon (;) identifies the end of a statement

For readability, we've shown the semicolon preceded by a space.

Our demo statement uses four SQL keywords: **SELECT**, **FROM**, **WHERE**, and **LIKE**.

It also uses two operators: the “all fields” operator (*) and the wildcard (%).

That leaves two variables—a table (**contact**) and a field in that table (**first**)—and one expression (**first LIKE '%jo%'**).

Beyond what we've shown you here, there are several SQL keywords. There are also many logical operators. There are even a handful of different wildcard operators. After all, SQL is a full-blown language.

In addition to basic queries, SQL operations can involve joining tables to create a combined record set, applying value-handling logic, imposing sorting and ordering of output, limiting output to a number of results, and creating “views,” which are essentially queries that have been saved for frequent reuse. Database theory goes further to include proper schema development, benchmark testing (identifying query bottlenecks), and optimization tactics, which are essential for massive, hardworking databases. However, those topics belong to a serious study of SQL techniques.

NOTE

While we encourage you to move beyond this appendix and into a good book discussing MySQL, you'll also find the MySQL website to be a significant reference: <http://dev.mysql.com/doc/>.

Similarly, W3Schools.com provides some SQL tutorials that might benefit you: <http://www.w3schools.com/sql/default.asp>.

MySQL Statements

In this section we consider the **SELECT**, **INSERT**, and **UPDATE** statements we use with the **ContactManager** in Chapters 16 and 17. It's important to realize that we'll be submitting these statements later via PHP as query strings, so while we're showing them here as pure SQL statements, they won't necessarily read so smoothly in PHP, which will require concatenating them.

The SELECT Statement and Fully Qualified References

Example B-1 shows the **SELECT** statement called by **ContactManager** that will be used to initially load as well as refresh the list of contacts whenever the database is changed.

Example B-1. MySQL *SELECT* statement

```
SELECT *
FROM `contacts`.`contact`;
```

But something's a little different from the queries we saw earlier. What's this syntax?

```
`contacts`.`contact`
```

It's a *fully qualified reference* identifying the database→table relationship targeted by the query. Notice backticks (`), *not* single quotes ('), are used to wrap both schema elements. The dot (.) denotes the table (**contact**) belongs to the database (**contacts**). Between phpMyAdmin and our PHP scripts, we won't need to use fully qualified references, but right now we're showing you pure SQL that relates to the functions of the **ContactManager**, so we'll continue using absolute references for the next couple of example statements.

Fully Qualified References

The SQL statement we began with is an example of an unqualified table reference:

```
SELECT *
FROM contact ;
```

There's nothing in the SQL to indicate which database contains the table "contact". As long as a database is already active, MySQL will infer that the requested table belongs to the active schema. By contrast, a fully qualified table reference explicitly states which database holds the desired table:

```
SELECT *
FROM `contacts`.`contact` ;
```

In a fully qualified reference, each schema element is wrapped in backticks (`) and joined by a dot (.) to indicate a relationship between the elements. Depending on your query vehicle (e.g., command line, GUI application, or code), you might not need to use a fully qualified table reference. At some point, though, you'll definitely need to activate a database. Activating a database via the command line requires the **USING** keyword. However, activating a database in some GUIs simply requires clicking on a database to select it.

Thanks to the handy PHP function `mysql_select_db()`, we won't need to use fully qualified references in our PHP scripts, which is nice because it will save us additional concatenation.

The INSERT Statement

The **INSERT** statement is used to add a new record to a table. So far we've seen several **SELECT** statements, but there are some big differences between **SELECT** and **INSERT** statements. Let's see our first **INSERT** statement in Example B-2.

Example B-2. MySQL INSERT statement

```
INSERT INTO `contacts`.`contact` (
  `type` ,
  `first` ,
  `last` ,
  `email` ,
  `phone`
)
VALUES (
  'person',
  'Molly',
  'Tovaklas',
  'mollytov@learningflex4.com',
  '1234567890'
);
```

This query has three parts. The first part fully qualifies the table to receive a new record. The second part of the statement declares which fields will receive values, and the third part assigns each field a value. The values in the third part should appear in the same order as the fields in the second part. Later, when we create this script in PHP, we won't need the absolute qualification or the backticks.

The UPDATE Statement and Primary Key Field

After seeing an **INSERT**, the **UPDATE** statement shouldn't look so scary. The following SQL submits changes to the record created in the previous statement. After a fully qualified reference, a **SET** clause contains field/value assignments. Note the fields are wrapped in backticks (`) and the values are wrapped in single quotes ('). The presence of the single quotes tells MySQL the value is a **String**. If you wanted to assign a numeric value (see the **WHERE** clause, which we discuss in a moment), you would assign the value directly, without quotes. Example B-3 shows the SQL.

Example B-3. MySQL UPDATE statement

```
UPDATE `contacts`.`contact`
SET
  `type` = 'person',
  `first` = 'Molly',
  `last` = 'Dogiklas',
  `email` = 'mollydog@learningflex4.com',
  `phone` = '1234567890'
WHERE `contact`.`id` = 16 ;
```

WARNING

In Example B-3, notice that the last field/value assignment is not followed by a comma. Adding an unnecessary comma after the last field/value assignment is a frequent sort of error among new SQL writers.

NOTE

*Perhaps you noticed that only the dog's last name and email changed between the **INSERT** and the **UPDATE** statements (yeah, Molly is a dog). If, going into the **UPDATE**, we knew we were changing only two values, we could have created a shorter **UPDATE** statement. To keep things simple, though, we chose to resubmit everything. This approach will make more sense when you see it applied to the **ContactManager** application in Chapter 16.*

The last part of this statement creates a **WHERE** clause to target the record we want to change. How you approach an **UPDATE** is significant because the condition could affect one, many, or every record in a table. In our case, we only want to modify a single, unique record. Our condition will use the table's *primary key* field. The database increments this field's value automatically, the field is guaranteed to have a value, and each record has a truly unique primary key value.

```
WHERE `contact`.`id` = 16 ;
```

That line tells our database the preceding **UPDATE** should apply only to the row that has an **id** of 16. We know there will be only one match, but the database has no reason to assume such. Therefore, as the developer writing the **UPDATE**, you have to be sure your update will apply as you intend.

Like the previous statement, this SQL uses an absolute reference and contains field names wrapped in backticks; neither will be necessary later in our PHP code.

That concludes our brief SQL language primer. In the next section, we use phpMyAdmin to create the **contacts** database and the **contact** table you'll need in Chapters 16 and 17.

Creating a Database with phpMyAdmin

Re-creating *contacts.xml* as a MySQL database will involve four steps:

1. Creating the **contacts** database
2. Creating the **contact** table
3. Activating six fields for the table (**id**, **type**, **first**, **last**, **email**, and **phone**)
4. Inserting some records

Creating the Database

To get going, start WAMP, open the WAMP menu, and launch phpMyAdmin. The “Create new database” utility is located near the middle of the screen. Supply **contacts** for the database name and click on Create (Figure B-1).

In the next view, you'll be prompted to add a table.

Creating the Table

Where you see “Create new table on database”, provide the name **contact** and enter **6** for the number of fields. Then click Go (Figure B-2).

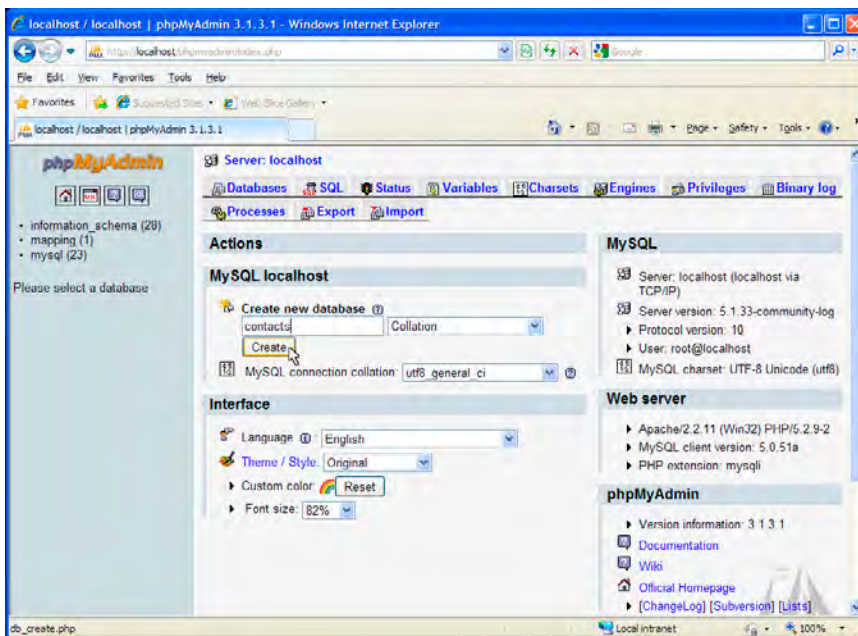


Figure B-1. Creating the contacts database using phpMyAdmin

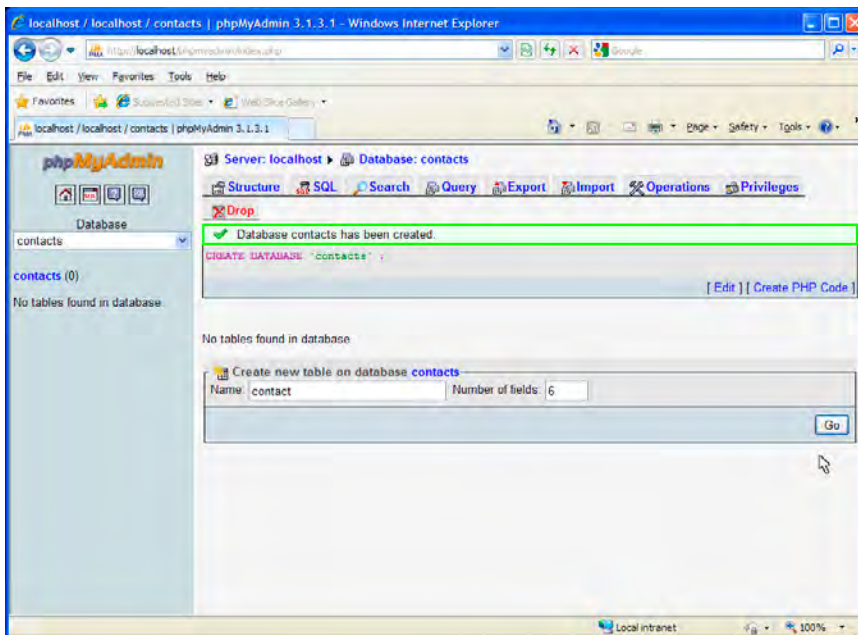


Figure B-2. Creating the contact table with six fields

As you submit commands using the GUI, notice that phpMyAdmin will output the SQL it builds to carry out your instructions. This is a handy feature that allows you to gradually absorb some SQL over time.

With the database and the table defined, now it's time to add the attribute fields.

Adding Fields

After creating the table, phpMyAdmin should have moved you ahead to add six fields. This is a rather wide page, but we're mostly interested in the far left and the far right.

While you're still far to the left, enter each field name and assign its data type. Working your way down the page, supply field name and data type relationships as indicated in Table B-1. Of course, the field names should be added to the input controls appearing below "field", and the data type values should be selected from the combo boxes below "type". For the **VARCHAR** data types, add an appropriate length/value assignment, as indicated by the numbers in parentheses in the table.

Table B-1. Schema for the contact table

Field name	Data type	Notes
id	INT	Primary key, auto-incrementing
type	VARCHAR (10)	Either "contact" or "business"
first	VARCHAR (25)	First name value
last	VARCHAR (25)	Last name value
email	VARCHAR (75)	Email value
phone	VARCHAR (10)	Phone value

Now scroll to the far right of the page. For the first row, which represents the **id** field, select PRIMARY for its "index" type and check the box corresponding to "A_I" (Figure B-3). These two settings establish **id** as the primary key and allow the database to perform auto-incrementing as new records are added. The combination of these two settings means each record will always have an **id** value, and that value will always be unique for that record. As stated in the previous section, our PHP **UPDATE** script will rely on the **id** field to target the proper record for modification.

With all the fields defined, scroll to the bottom of the page and select Save. That completes the task of building the database schema. Now we need to add some records so there will be information to pull when we use PHP to make requests.

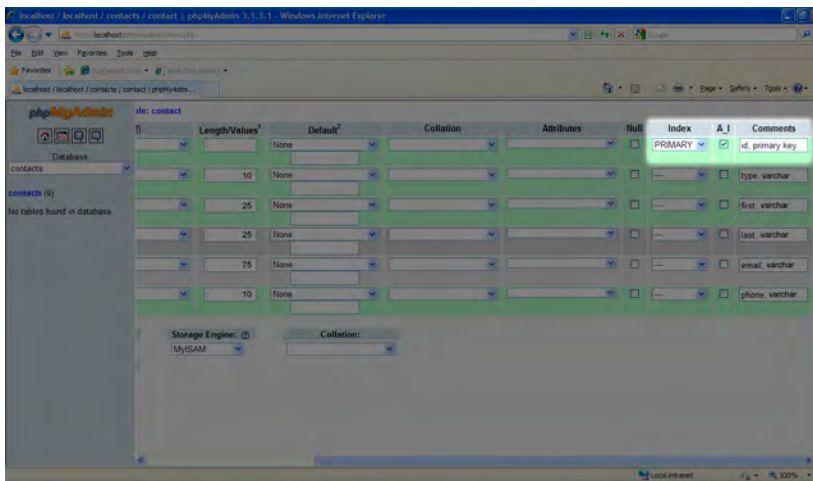


Figure B-3. Enabling primary key and auto-increment to the `id` field in the `contact` table

Inserting Records

It's easy to add table records using phpMyAdmin. When you emerge from defining fields, select **Insert**, which is among the navigation tabs. In the **Insert** view, scroll down the page and add values for any contact records you want added to the table. One caveat: leave the `id` field blank! Remember, the database will automatically add this value every time a record is submitted, so we don't want to interfere with that process.

Create at least a few contact records before proceeding (Figure B-4).

Browsing the Data

Testing your scripts directly against the database before ever using them in your applications is always a smart move. This particularly applies to long, dense scripts, but it also makes sense for us because we're learning. Testing your SQL first allows you to proceed with confidence, but sometimes it will reveal problems with the SQL syntax that you'll need to fix.

Assuming you've just finished adding some records to the `contact` table in phpMyAdmin, you can test SQL statements by navigating to the "SQL" tab and entering some SQL. For example, navigate to the SQL view, type the following query (which you're no doubt getting tired of seeing), and then click **Go** (Figure B-5):

```
SELECT * FROM contact ;
```

WARNING

As you insert records into the `contact` table, don't add values to the `id` field! The database will provide this value automatically.

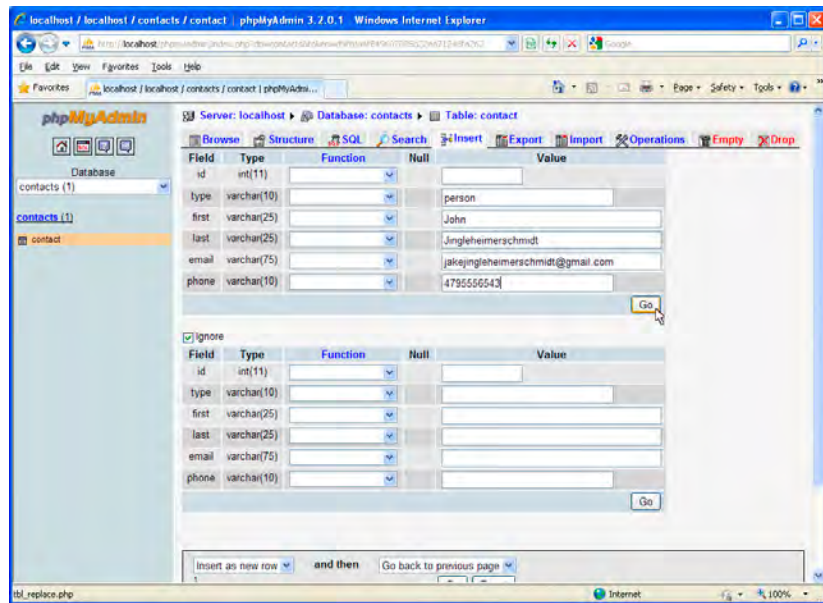


Figure B-4. Leaving the id field empty while adding records to the contact table

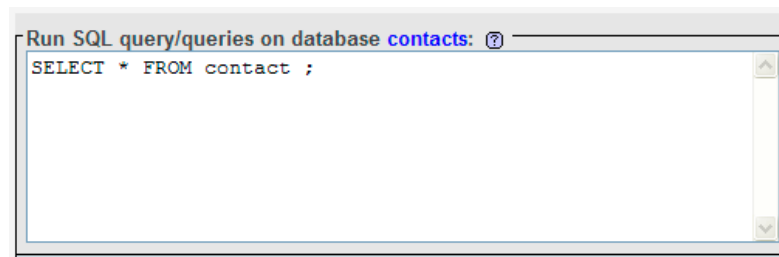


Figure B-5. Preparing to submit an SQL query via phpMyAdmin

You should receive a response almost immediately showing all the records of the **contact** table (Figure B-6).

If you're inclined, you're welcome to test the **INSERT** and **UPDATE** statements while you're still in phpMyAdmin; however, now that your **contacts** database and **contact** table are established, we'll conclude Appendix B.

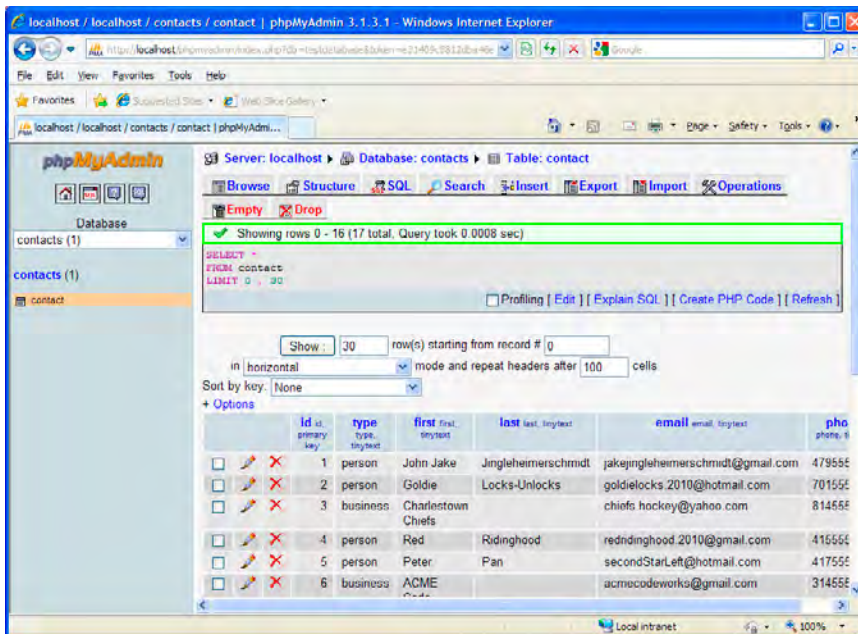


Figure B-6. Reviewing the MySQL response

NOTE

Actually, *phpMyAdmin* will impose a **LIMIT** clause with a condition of 30 to prevent a potentially unfortunate query that might attempt to load a 15,000-record response right in the web browser. That's why your query was revised to look like this:

```
SELECT *
FROM contact
LIMIT 0, 30
```

Summary

In this appendix, we covered the basic tenets of SQL scripting in MySQL.

The first half of the appendix gave you a feel for the syntax and language elements going into SQL statements. You learned about the **SELECT**, **INSERT**, and **UPDATE** statements, and you learned how to zero in on specific records using the **WHERE** clause. We also discussed a few schema basics, including fully qualified queries, the primary key, and auto-incrementing index values.

In the second half of the appendix, you learned how to use *phpMyAdmin*, which comes with a WAMP/MAMP installation, and how to create a database, add a table, build a table schema, and insert records into the table. You even learned how you could use *phpMyAdmin* to test SQL statements before you try to use them in your applications.

Appendix C provides a similar primer on PHP, including a quick tour of the basic language elements and syntax, as well as a discussion of the scripts you'll need for the **ContactManager** application in Chapters 16 and 17.

PHP BASICS

PHP is a mature and powerful server-side scripting language, but our shallow usage of it in this book is concerned with one simple goal: bridging a Flex 4 application to a MySQL database. As such, the PHP files we present are mere outboard functions, stored on the server and waiting for the **ContactManager** application created in Chapters 16 and 17 to call them to either retrieve data from or submit data to a MySQL database.

The following PHP primer addresses PHP with our immediate goals in mind. First, we cover PHP basics, including the structure of a PHP file, how to declare variables, how to concatenate strings, and the ever-essential PHP MySQL functions. Once you're familiar with the very, very basics, we discuss the actual PHP scripts we'll use with the **ContactManager** application from Chapters 16 and 17.

IN THIS CHAPTER

Language Elements and
Syntax

The PHP Scripts
Summary

Language Elements and Syntax

The first section of our PHP primer is concerned with the anatomy of a PHP file and the scripting essentials you'll need to create and understand the scripts used by the **ContactManager** application.

PHP Files

PHP files should be appropriately named and end with the *.php* extension. Although PHP files can be edited using any code editor, in Appendix A we demonstrated how to add the PHP Development Tools (PDT) plug-in to your Flash Builder installation, so we expect you'll use Flash Builder to create and edit PHP files. Within the PHP file, code is wrapped with the following opening (`<?php`) and closing (`?>`) tags:

```
<?php  
    // PHP code goes here  
?>
```

As demonstrated in this example, you can use double slashes (`//`) to add comments within PHP scripts. You can also use the familiar comment block system (`/* */`) to isolate blocks of comments within PHP.

Variable Declaration in PHP

We'll certainly be working with some variables. In PHP, variable declaration begins with a dollar sign (`$`) and continues with the variable name, like this:

```
$myString = "this is my string";
```

Notice the assignment operator (`=`) applies a value to the variable, and the expression is closed by a semicolon (`;`).

If you wanted to create a numeric variable you would *not* use quotes:

```
$myNumber = 30;
```

Unlike with ActionScript, PHP variables are *loosely typed*, meaning the appropriate data type is interpreted on the fly by the PHP server. Using loosely typed variables may feel different as you get started with PHP, but as with any programming language, after a short while you'll become comfortable with the differences.

Concatenating Strings in PHP

The period (`.`) has a special purpose in PHP. Specifically, it's called the *concatenation operator*, and it's used to add one string expression to another. In the following example, two strings (`$myString` and `$myAge`) are added together using the concatenation operator:

```
<?php

$myString = "this is my age: ";

$myAge = 30;

print $myString . " " . $myAge;

?>
```

The **print** function has the purpose of dispatching output from a PHP script. It has a companion function, **echo**, and you'll see them both used off and on. For the most part, however, these functions are interchangeable and their usage comes down to developer preference.

If you saved this code as a PHP file, copied it to a directory in the web scope, and then requested it using a web browser, you'd receive the following output in your browser:

this is my age: 30

PHP MySQL Functions

When connecting to a MySQL server from PHP, you'll need to know the MySQL server's IP address, port number, and a valid username and password. For a default MySQL installation, these respective values would be:

- IP: 127.0.0.1 (aka the *localhost*)
- Port: 3306
- User: root
- Pass: (empty string)

However, if you're connecting to a MySQL server maintained by your web host, you might need to reference the server using its *public IP* (i.e., its externally facing Internet network address) and the username/password combo determined when you created your web hosting account.

Of course, to be of any use, a MySQL installation needs at least one database having one table. In fact, that describes the database schema we created in Appendix B, and the one we'll use for the **ContactManager** application—one database named **contacts** and one table named **contact**.

Keep this information in mind for a moment while we look at the four PHP MySQL functions we'll be using. For each PHP MySQL function we state its name, show how it should be used relative to our first PHP script (which appears in the next section), and briefly discuss its purpose:

`mysql_connect()`

```
$link = mysql_connect("127.0.0.1", "root", "");
```

This function establishes a connection to the MySQL server and stores it as a variable (**\$link**). The function takes the IP of the server (*localhost* in this case) and a valid username and password (the default root user in this case).

`mysql_select_db()`

```
mysql_select_db("contacts", $link);
```

A MySQL installation can support many databases. In order to identify a database to query, it must be established as the active database. This function takes the name of a database (**contacts**) and makes it active for a connection (**\$link**).

`mysql_query()`

```
$result = mysql_query($query, $link);
```

The query function takes two variables: an SQL statement as a **String** (**\$query**), and an instance of a connection (**\$link**). This function returns the product of the MySQL query as an array, which is stored in a variable (**\$result**).

NOTE

The root password for a default MySQL install is truly an empty string. If you connect to a new MySQL installation using the command line, simply hit Enter/Return when prompted for a password. If you're connecting in script, as you will with PHP, you'll need to explicitly declare an empty string (""). If you use phpMyAdmin to connect to a default MySQL installation—which we demonstrated in Appendix B—it won't prompt you for a password.

mysql_fetch_array()

```
$row = mysql_fetch_array($result)
```

The last PHP function we'll use takes the array created by `mysql_query($result)`, accesses the first row of data in that result, and stores it as another array (`$row`). If `mysql_fetch_array` is wrapped in a loop, it will continue advancing to the next record in the original `$result` array after each successful pass. This means we can use it to cycle through each record of a MySQL query response.

NOTE

An SQL query is a statement that prompts a database to return a data array based on a special request, but it can also be used to add, change, or delete data. This is an example of a simple SQL query:

```
SELECT * FROM contact;
```

This statement would return every record from the table named "contact". If we created this query as a String variable in PHP, it would look like this:

```
$query = "SELECT * FROM contact";
```

Notice that we didn't include the SQL semicolon (;) in the PHP query string? That's because PHP's `mysql_query` function automatically terminates the SQL statement, which is otherwise accomplished using the semicolon.

That concludes our PHP primer. In the next section we consider the scripts we'll use with the **ContactManager** application in Chapters 16 and 17.

The PHP Scripts

In this section we consider three PHP scripts. You won't create the scripts here; rather, you'll create them in Chapter 16. However, in this section we provide a breakdown and discussion of the scripts used by the **ContactManager** application. So, you might want to progress through Chapter 16 and create the files when they're called for, and then if something is unclear, return to this section for more consideration.

The first script, *loadContacts.php*, handles the task of querying the MySQL **contact** table and returning the result as an XML-formatted string. The second script, *insertContact.php*, receives **POST** variables from the calling application and repackages them as a new record for the **contact** table using a MySQL **INSERT** statement. Finally, the third script, *updateContact.php*, receives **POST** variables and uses a MySQL **UPDATE** statement with a **WHERE** clause to modify a specific record of the **contact** table.

The first script requires some dense concatenation and uses a PHP **while** loop. It will certainly be the hardest script to grasp. As such, we chose to discuss it first and spend the majority of this appendix dissecting it. By the time you're comfortable with the first script, the latter scripts will be much smoother.

NOTE

If you want to do some extra reading and/or review of the PHP MySQL functions, check out the material posted at [w3schools.com](http://www.w3schools.com). You can find a lot of great information on this site, including topics other than PHP and MySQL. It's definitely one to bookmark: http://www.w3schools.com/PHP/php_ref_mysql.asp.

loadContacts.php

As stated at the top of this section, *loadContacts.php* will be our crown jewel of PHP development. So prepare for a headache. Just kidding, it won't be that bad.

The block of PHP code in Example C-1 has three parts. The first part connects to the MySQL database and performs a query. The second part conducts an intense concatenation on the MySQL response to create XML output. Finally, the last part returns an XML-formatted string as the product of the script.

We'll handle our discussion after the code.

Example C-1. *loadContacts.php*

```
<?php
    header("content-type: text/xml");

    // development link:
    $link = mysql_connect("127.0.0.1", "root", "");
    mysql_select_db("contacts", $link);

    // query statement
    $query = "SELECT * FROM contact";
    $result = mysql_query($query, $link);

    $xmlData = "";
    $xmlData = "<contacts>\n";
    while($row = mysql_fetch_array($result))
    {
        $xmlData .= " <contact id='" . $row['id'] . "' type='" .
        $row['type'] . "'>\n";
        $xmlData .= " <firstName>" . $row['first'] . "</firstName>\n";
        $xmlData .= " <lastName>" . $row['last'] . "</lastName>\n";
        $xmlData .= " <email>" . $row['email'] . "</email>\n";
        $xmlData .= " <phone>" . $row['phone'] . "</phone>\n";
        $xmlData .= " </contact>\n";
    }
    $xmlData .= "</contacts>\n";

    // return the XML string
    echo $xmlData;
?>
```

The first line of the file uses PHP's **header()** function to define the output format:

```
header("content-type: text/xml");
```

The next several lines perform the MySQL connection, database activation, and query. When we introduced PHP MySQL functions in the previous section, we used several lines corresponding to this code block. So hopefully these MySQL functions look recognizable. Our real work begins at this line:

```
$xmlData = "<contacts>\n";
```

That line begins our XML output, and the `\n` character sequence signals a line break. The next line starts a **while** loop around the `mysql_fetch_array()` function:

```
while($row = mysql_fetch_array($result))
{
```

As long as the function can continue reading unique records in the response array, `$result`, the **while** loop will continue cycling. Each pass through the loop adds a new record to the XML output. Once `mysql_fetch_array()` fails to cycle, the loop will terminate.

Now we'll present a significant discussion concerning the first line of the **while** loop. The next three paragraphs discuss qualities of the first line, and then the script will be reprinted with the discussed portions emphasized.

First, note the *concatenation-assignment* (`.=`) operator, which tells PHP to append, rather than overwrite, the expression following the operator to the growing `$xmlData` string. This script makes heavy use of this operator within the **while** loop:

```
$xmlData .= " <contact id='" . $row['id'] . "' type='" . $row['type'] . "'>\n";
```

Next, notice how the first line builds the opening `<contact>` tag, including the **id** and **type** attributes. Because each string expression is wrapped in quotes (`"`), we have to use single quotes (`'`) to distinguish attribute values within the XML as well as identify fields/values we're extracting from the MySQL `$row` array. The last tidbit—`">\n";`—is cryptic; however, it simply closes the final attribute assignment with a single quote (`'`), adds the tag-closing right-angle bracket (`>`), and finishes with a line break (`\n`). The semicolon (`;`) terminates the line of PHP. These portions are emphasized in the following reprint of the first line in the **while** loop:

```
$xmlData .= " <contact id='" . $row['id'] . "' type='" . $row['type'] . "'>\n";
```

Finally, values coming out of the MySQL response are accessed using the syntax `$row['fieldname']`, where `$row` is a PHP variable representing one record from the full MySQL response, and where `'fieldname'` is a column value from that record. This reprint emphasizes the MySQL values going into the first line of PHP:

```
$xmlData .= " <contact id='" . $row['id'] . "' type='" . $row['type'] . "'>\n";
```

Therefore, `$row['id']` represents the **id** field from a contact record, and that value is added to the XML string via the concatenation operator. Similarly, `$row['type']` returns the **type** value.

To summarize the script, we're creating XML programmatically by concatenating several strings and variable values together while accounting for XML syntax rules, thus making one long, XML-formatted string.

Having discussed the concatenation, the first line of the loop might produce output like the following:

```
<contact id= '1' type= 'person'>
```

Because it doesn't involve attributes, the second line of the loop is much easier to envision as XML:

```
$xmlData .= " <firstName>" . $row['first'] . "</firstName>\n";
```

It's fairly easy to imagine the second line rendering as follows:

```
<firstName>John</firstName>
```

The remaining lines of the **while** loop use the same approach to populate the other contact details, so it is unnecessary to explain them one by one.

The final line of the PHP script uses the **print** function to return the XML output to the requesting application (replacing **print** with **echo** would work equally well):

```
print $xmlData;
```

insertContact.php

The purpose of *insertContact.php* (shown in Example C-2) is relaying several passed-in **POST** values to the MySQL database in the form of a new contact record.

To break down the sequence, the script begins by receiving several inbound values, wrapped together in an array (**\$_POST**). These **POST** variables are then separated out of the array into individual variables. Next, an **INSERT** statement is concatenated together within the body of a **mysql_query()** function. Note that the MySQL link variable (**\$link**) is the final parameter of the **mysql_query()** function.

Example C-2. *insertContact.php*

```
<?php
// development link:
$link = mysql_connect("127.0.0.1", "root", "");
mysql_select_db("contacts", $link);

// incoming variables
$type = $_POST['type'];
$first = $_POST['first'];
$last = $_POST['last'];
$email = $_POST['email'];
$phone = $_POST['phone'];

// query statement
mysql_query
(
```

NOTE

*We address the deeper meaning of **POST** in Chapter 16, where we build the **ContactManager** application. Look for the description in the sidebar titled “HTTP Request Methods” on page 363.*

```

        "INSERT INTO contact
        (
            type,
            first,
            last,
            email,
            phone
        )
        VALUES
        (
            '" . $type . "',
            '" . $first . "',
            '" . $last . "',
            '" . $email . "',
            '" . $phone . "'
        )",
        $link
    );
?>

```

updateContact.php

Like the **INSERT** script, the **UPDATE** script in Example C-3 also receives values from Flex via a **POST**. First we set up the MySQL connection, and then we filter out the **POST** variables and concatenate them into a query string within a `mysql_query()` function. The major difference concerning this script is its inclusion of the **id** value (`$id`) and using it to target a specific record via the SQL **WHERE** clause. Also, yet again, notice that the MySQL link variable (`$link`) is the final parameter of the `mysql_query()` function.

Example C-3. *updateContact.php*

```

<?php
// development link:
$link = mysql_connect("127.0.0.1", "root", "");
mysql_select_db("contacts", $link);

// incoming variables
$id = $_POST['id'];
$type = $_POST['type'];
$first = $_POST['first'];
$last = $_POST['last'];
$email = $_POST['email'];
$phone = $_POST['phone'];

// query statement
mysql_query
(
    "UPDATE contact SET " .
    " type = '" . $type .
    "', first = '" . $first .
    "', last = '" . $last .
    "', email = '" . $email .
    "', phone = '" . $phone .
    "' WHERE id = " . $id
    , $link
);
?>

```

Summary

This appendix introduced you to the very, very basics of PHP scripting.

In the first half of the appendix you learned about the anatomy of a PHP file, and we discussed how to declare variables and concatenate strings in PHP. We presented the PHP MySQL functions, and you learned about the parameters used by the MySQL functions to access a database and create a link.

In the second half of the appendix, we provided a discussion and breakdown of the three scripts used by the **ContactManager** application in Chapters 16 and 17. We paid special attention to the techniques used to stage a MySQL query, loop through the result, and build an XML-formatted string response. Finally, we discussed the simpler scripts used to submit data to MySQL, including the **INSERT** statement, as well as an **UPDATE** statement with a **WHERE** clause.

COMPILING FLEX APPLICATIONS ON LINUX USING THE COMMAND LINE

Contributed by Matthew Sabath

In this appendix, we discuss the steps necessary to compile a Flex 4 project on a Linux system using Adobe's free command-line compiler. Our example uses the Ubuntu 10.04 distribution with a default installation.

This tutorial walks you through the following processes:

- Installing Flash Player 10
- Installing Java
- Downloading the Flex 4 SDK
- Creating a workspace and a project folder structure
- Adding a simple Flex MXML application file
- Adding environment variables to the Linux shell
- Arranging a project's *flex-config.xml* file.
- Creating a reusable Bash script to call the compiler

IN THIS CHAPTER

- Install Flash Player 10
- Install Java
- Download the Flex 4 SDK
- Create a Project Folder Structure
- Add an MXML File
- Add Environment Variables
- Tweak the Project Configuration File
- Create a Reusable Compiler Script in Bash
- Compile and Test
- Summary

Install Flash Player 10

Before you can run Flex 4 applications on Linux, you need to install Flash Player 10. This can be accomplished using the Ubuntu Software Center.

Access Ubuntu Software Center by opening the Applications menu and selecting Ubuntu Software Center (Figure D-1).

WARNING

Some Linux users have experienced difficulties installing Flash Player 10 over the top of Flash Player 9. If you run into trouble later getting Flex 4 applications to run in your Linux installation—or if you want to make sure you're starting with a clean slate before even risking frustration—uninstall Flash Player 9 like so:

1. Open the Places menu and select Home Folder.
2. In the Home directory, open the View menu and enable Show Hidden Files.
3. Find the folder named `.adobe` and open it.
4. In the `.adobe` folder, delete the folder named `Flash_Player`.

By following these steps, you can ensure older versions of the Flash client will not create issues for your installation of Flash Player 10.

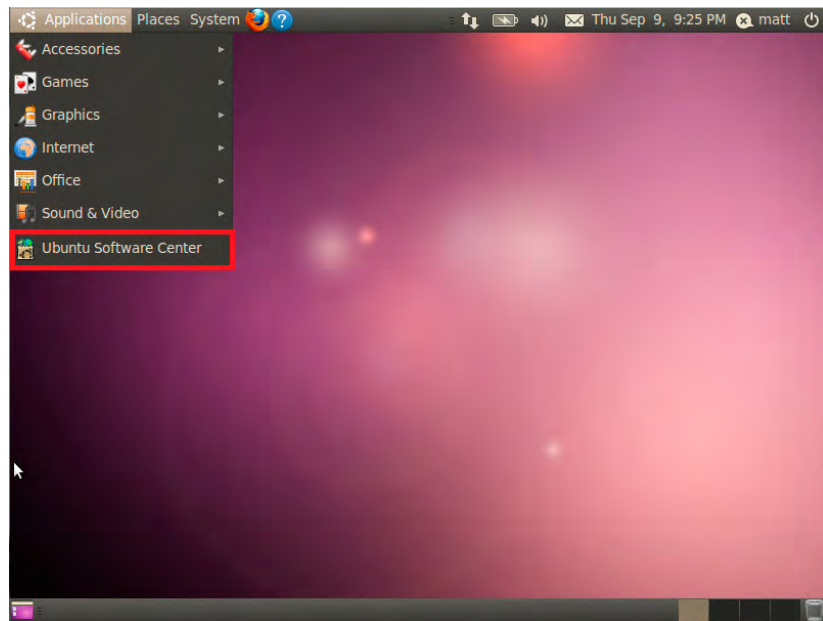


Figure D-1. Launching Ubuntu Software Center

Next, type **Adobe 10** in the search box; then, select Adobe and click on More Info (Figure D-2).

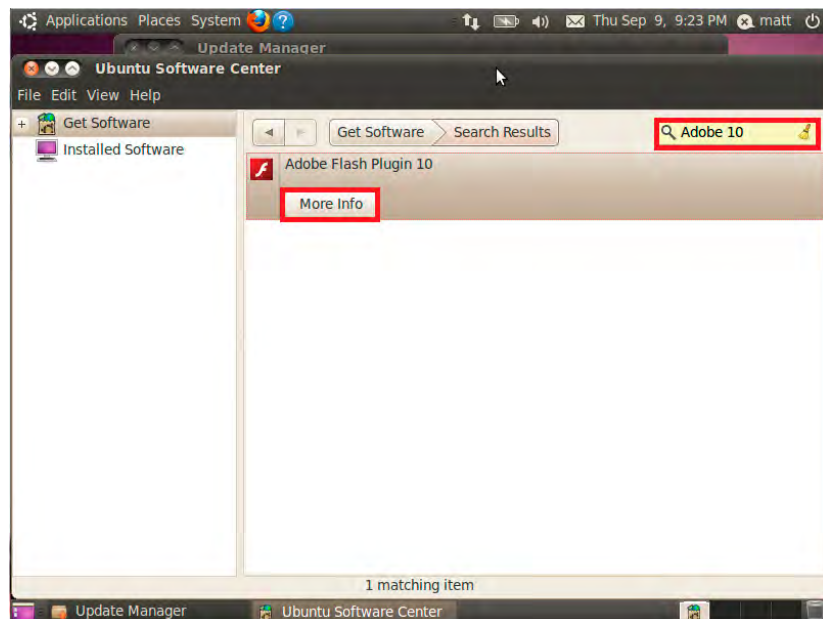


Figure D-2. Selecting Adobe Flash Plugin 10 for installation

You should be presented with a button noting “Use This Source”. Click it, and then select Install (Figure D-3).



Figure D-3. Installing Adobe Flash Player 10

That should complete the Flash Player 10 installation. At this point, it would be wise to visit a Flash-based website to ensure the plug-in installed correctly.

Install Java

Next, we'll install Java. For this step, you need to install software from the command line. Fortunately, this is a simple process using **apt-get**. Just open a command terminal (Applications→Accessories→Terminal) and enter the following lines (be sure to preserve spaces between quotes):

```
sudo add-apt-repository "deb http://archive.canonical.com/ lucid partner"
sudo apt-get update
sudo apt-get install sun-java6-jre sun-java6-plugin sun-java6-fonts
```

Once you're finished, test the installation by entering the following command into the terminal. This command should report the version number of Java currently running on your system:

```
Java -version
```

Download the Flex 4 SDK

Adobe's Flex 4 SDK needs to have access to the Flex framework libraries and the command line compiling utility. At the time of this writing, the following link provided various downloads of the Flex 4 SDK: <http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4>.

For this example, we used the SDK milestone corresponding to Flex 4.1 Update under the column Adobe Flex SDK (Download ZIP, 169 MB).

Go ahead and download the SDK ZIP. When the file finishes downloading, move it (right-click→Cut) to your home folder (Places→Home Folder).

Extract the archive in your home folder (right-click→Extract Here); see Figure D-4.

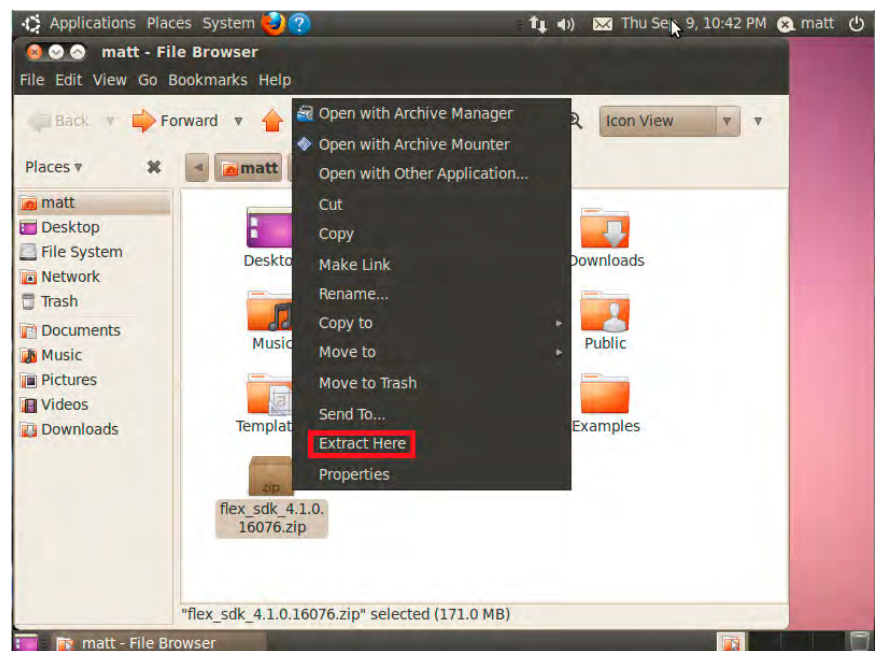


Figure D-4. Extracting the Flex 4 SDK to the Home Folder

To complete the step, rename the extracted directory *flex41*.

Create a Project Folder Structure

You need to build a workspace and a project folder for your application. The project folder will hold the project's packages, source code, and any supporting files. For convenience, we'll create the workspace and project folder within our Home Folder directory.

Open the Home Folder directory and create a folder with the name *flex41wk*:

Right-click→Create Folder: *flex41wk*

Now browse into the *flex41wk* folder and create a project folder named *hello*:

Right-click→Create Folder: *hello*

Within the project folder, create two more folders named *src* and *bin*:

Right-click→Create Folder: *src*

Right-click→Create Folder: *bin*

Figure D-5 shows the result.

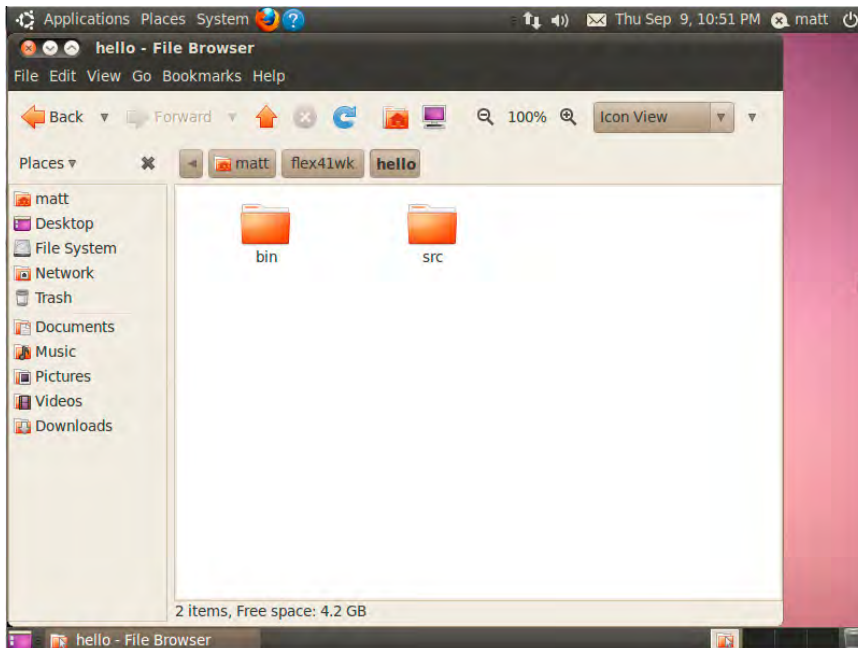


Figure D-5. Showing the project's folder structure

Add an MXML File

With a new workspace and a project folder structure, you're ready to start adding some Flex code. We'll keep the example simple by adding a single MXML file. Browse into your *src* folder and create a new file called *hello.mxml* (right-click→Create Document→Empty file). Double-click *hello.mxml* to open it in a text editor, and then add the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/mx">

    <s:HGroup horizontalCenter="0" verticalCenter="0">

        <s:TextInput id="inputTI"/>

        <s:Button label="Say Hi!"
            click="{outputTI.text=inputTI.text; inputTI.text=''}"/>

        <s:TextInput id="outputTI"/>

    </s:HGroup>

</s:Application>
```

This is some pretty simple Flex code. We have two **TextInput** controls and a **Button**. The **Button** control's **click** event actually contains two inline expressions. The first expression simply moves the contents of the first **TextInput** (**inputTI**) into the second **TextInput** (**outputTI**). The second expression follows the first expression by clearing the contents of the first **TextInput**.

Add Environment Variables

Environment variables work just like variables in programming—once you establish them, you can easily recall them using command-line functions. In Linux, the *.bashrc* file is referenced by the Bash shell. This file determines the behavior of interactive shells, and it also contains environment variables. In this section, we'll add some environment variables to the *.bashrc* file. To get started, open a terminal shell (Applications→Accessories→Terminal).

When the terminal opens, type:

```
gedit ~/.bashrc
```

This opens the *.bashrc* file for editing. For the purposes of this example, don't worry about the contents of this file; just scroll to the bottom of the document and add the following lines:

```
#full path to flex4 sdk bin added to PATH variable
PATH=$PATH:~/flex41/bin/
export PATH
```



```
#full path to flex4 sdk frameworks  
flexlib=~/.flex41/frameworks  
export flexlib
```

When you are done, save the changes and close the file. In your terminal window, enter the following command to refresh your environment and include the new variables:

```
source ~/.bashrc
```

You should now be able to type `echo $flexlib` to confirm that your environment shell is updated (Figure D-6).

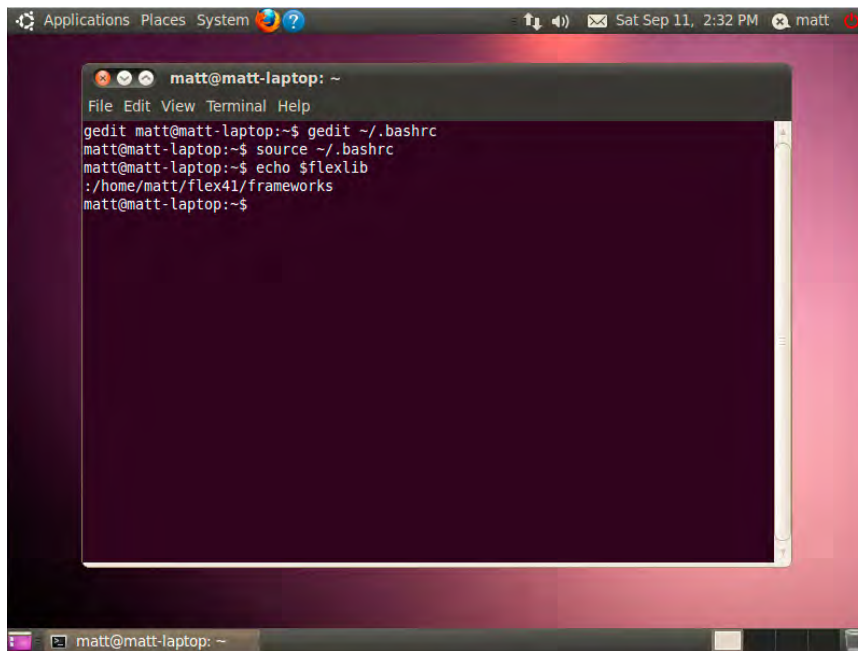


Figure D-6. Confirming the addition of environment variables

Tweak the Project Configuration File

The Flex 4 compiler parses an XML configuration file to glean information about the installation path of your Flex 4 SDK as well as the path to your project's `src` directory. Because a template of the Flex 4 configuration file is included with the SDK, we can copy it to the workspace's project folder and then modify it.

First, copy the configuration file from the SDK to your project folder like so (notice we're taking advantage of the `$flexlib` variable to shorten the command):

```
cp $flexlib/flex-config.xml ~/flex41wk/hello/flex-config.xml
```

Now we need to edit the configuration file so it will recognize the SDK location. Specifically, you'll modify the source path element in the *flex-config.xml* file (Figure D-7). It will be near the top of the file, and it needs to be a fully qualified address. Of course, you'll want to replace the profile name shown here (*matt*) with your profile name:

/home/matt/flex41wk/hello/src/

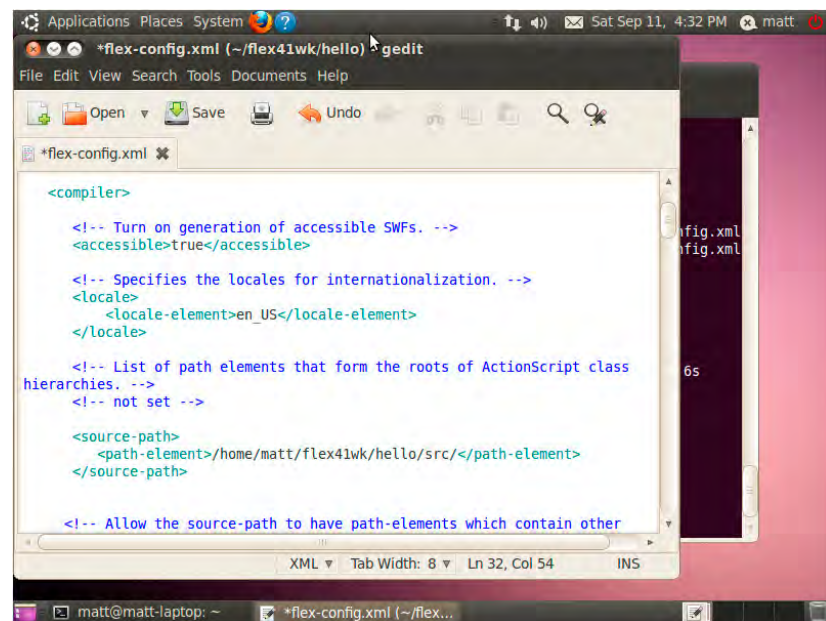


Figure D-7. Setting the source path to the project's `src` folder in the *flex-config* file

Now we need to set the paths for the libraries that the compiler will use. There are quite a few places in the *config* file where you'll need to alter the path. Everywhere the *config* file references a local path for a file, you need to add `${flexlib}` to the front of it. For example:

```
<external-library-path>
  <path-element>libs/player/{targetPlayerMajorVersion}.
    {targetPlayerMinorVersion}/playerglobal.swc</path-element>
</external-library-path>
```

becomes:

```
<external-library-path>
  <path-element>${flexlib}/libs/player/{targetPlayerMajorVersion}.
    {targetPlayerMinorVersion}/playerglobal.swc</path-element>
</external-library-path>
```

WARNING

The `<path-element></path-element>` node appearing in both of these code snippets should be a single line of code in your editor. Take caution to avoid a line break between `{targetPlayerMajorVersion}` and `{targetPlayerMinorVersion}`.

If you might benefit from a comparison, we provided our configuration file on the companion website. This may make it easier for you to ensure you changed all the necessary paths. The link is: <http://www.learningflex4.com/blogassets/flex-config.xml>.

Once you are finished editing, remember to save and close the file.

Create a Reusable Compiler Script in Bash

Next, we'll make a reusable Bash script to call the Flex compiler using settings specific to the project. For this we will create a new file and edit it.

Browse to Places→Home Folder→*flex41wk*→*hello*. Then, right-click the *hello* folder and select Create Document→empty file (Figure D-8).

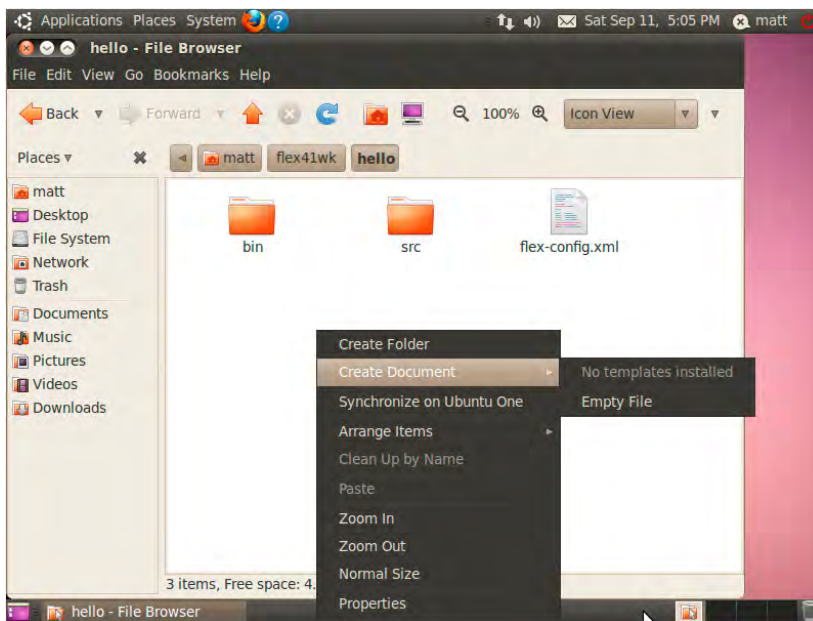


Figure D-8. Creating a new, empty file in the project folder

When the new file appears, name it *compile* (leave the extension blank).

Double-click the file to open it, and add the following lines:

```
#!/bin/bash
mxmclc -load-config flex-config.xml ./src/hello.mxml -output ./bin/hello.swf
```

Save the file. This Bash script handles the task of launching the compiler, passing it a reference to the *flex-config.xml* file, identifying the main application file, and specifying the output, which in this case is *bin/hello.swf*.

Finally, we need to change the permissions of our compile script to make it executable:

```
cd ~/flex41wk/hello
chmod 755 compile
```

Compile and Test

You should now be ready to compile and test. Compile the application using the following commands in the terminal window:

```
cd ~/flex41wk/hello
./compile
```

Assuming you don't receive any errors, browse to your compiled SWF file:

Places→Home Folder→flex41wk→hello→bin

In this directory, right-click on *hello.swf* and select “Open with Firefox”.

Summary

After completing this tutorial, you should be equipped with the knowledge to begin compiling Adobe Flex 4 applications using the command line on Ubuntu Linux.

The core phases of the exercise demonstrated how to set up an Ubuntu system with the Flash Player 10 plug-in, Java, the Adobe Flex 4 SDK, and special environment variables that simplify referencing the SDK components using terminal commands.

Additionally, we provided instructions for building Flex 4 applications. These steps included creating a proper project folder structure, knowing where to put the project's source code, arranging a project's *flex-config.xml* file, and implementing a reusable Bash script to simplify calling the compiler—a step that's most effective when the project is being compiled over and over again at some frequency.

INDEX

Symbols

\$ (dollar sign), 198
 += (addition assignment operator), 196
 <> (angle brackets), 56, 67, 317
 \ (backslash), 142, 197
 [Bindable] meta tag, 139
 {} (braces). *See* braces {}
 [] (brackets), 286
 : (colon), 74, 76, 318
 , (comma), 198
 . (dot), 319
 // (double forward slash), 87
 " (double quotes), 65, 66
 "" (empty string), 68
 = (equals sign), 66
 ! (exclamation mark), 283
 && (logical AND operator), 309
 || (logical OR operator), 310
 \n (new line), 83
 () (parentheses), 68, 72, 76
 | (pipe), 43, 318, 319
 # (pound symbol), 199
 ? (question mark), 330
 \r (carriage return), 83
 ; (semicolon), 65, 74, 318
 ' (single quotes). *See* single quotes (')
 _ (underscore), 114

A

abbreviations (naming conventions), 64
 Absolute/Basic layout, 153, 175
 absolute positioning, 153, 173
 absolute URLs, 227
 access modifiers, defined, 71

Accordion navigator
 Bounce effect and, 254–256
 functionality, 43, 254–256
 ACE (Adobe Certified Expert), 393
 action effects, 302
 Action Message Format (AMF), 8
 ActionScript. *See also* inline
 ActionScript
 assignment and concatenation, 66
 basic code block, 2
 class support, 79–84
 Collision! Flash game, 120–136
 commenting code, 87
 common events, 104–105
 controlling ViewStack with, 250–254
 data type support, 75–78
 dot notation, 64, 86
 effect support, 296
 functionality, 2, 63
 function support, 66–74
 handling events, 113–120
 listening for events, 109–120
 object support, 78–79
 operator support, 66
 relationship with MXML, 85–87
 researching events, 105–109
 responding to events, 109–120
 understanding events, 104
 variable support, 74
 viewing MXML generated code, 85
 ActionScript 3 API, 231, 264
 AddAction effect, 302
 addElement() method, 158, 159
 addEventListener() method, 87, 114, 116
 Add Folder dialog, 80
 addition assignment operator (+=), 196
 addition operator, 66, 142

Adobe Certified Expert (ACE), 393
 Adobe Flash Player. *See* Flash Player
 Adobe Illustrator, 288
 AdvancedDataGrid control, 210
 AIR (Adobe Integrated Runtime)
 ContactManager application,
 345–364
 creating installer, 376
 functionality, 4, 27
 FXG editors, 6
 installation files, 382
 WindowedApplication tag, 53
 Ajax, 9, 346
 Alert class
 functionality, 117
 show() method, 193, 196
 AlivePDF API, 388
 allowMultipleSelection property, 230
 ALPHA format (masks), 287
 alpha property, 325
 AMF (Action Message Format), 8
 angle brackets <>, 56, 67, 317
 angleYFrom property, 293
 angleYTo property, 293
 AnimateColor effect, 287
 Animate effect, 287
 animations
 events triggering, 286
 skinning support, 340
 ANT compiler, 17
 APIs (third-party), 388–389
 Appearance pane (Flash Builder), 34
 applicationComplete event
 Application container, 116, 118, 260
 external data and, 224
 functionality, 104, 105

Application container

- absolute positioning, 153
- applicationComplete event, 116
- applicationComplete handler, 118
- creating forms, 168
- creationComplete event, 115
- currentState property, 277, 281
- defaultButton property, 233
- functionality, 41, 53
- nested layouts and, 154
- selecting, 35

application layouts

- alignment considerations, 171–172
- constraints-based, 173–176
- container considerations, 165–168
- controlling whitespace, 162–165
- data bindings in, 143
- Display List, 155–160
- Flex layout process, 164
- padding, 162
- passing styles to children, 141
- planning, 169
- resizable, 143–144
- sizing considerations, 160–162
- spacers and lines, 168–170
- types of, 152–154
- visualizing structure, 171
- whitespace between containers, 149

Application tag

- Flex applications and, 53
- Flex layout process and, 169
- function placement and, 67
- <fx:binding/> tag placement, 146

Array class

- Flex 3 support, 213
- List-based components and, 215

ArrayCollection class

- lists of simple data, 210–213
- ViewStack and, 251

ArrayList class, 210

as3corelib library, 388

as3flickrlib library, 388

as3syndicationlib library, 388

ASDoc pane, 107

assets, embedding, 332

assignment

- data type, 74
- setting property values, 66

assignment operators, 66, 196

asSQL library, 388

Astra Web API, 231, 238, 389

attributes. *See also* properties

- adding to tags, 85
- as event listeners, 87
- as properties, 85
- style, 86, 315

audio player widget, 217

autoCenterTransform property, 293

B

backgroundColor property, 323, 369

backgroundStyle property, 323

backslash (\), 142, 197

Bak, Robert, 391

Bash scripts, 439

BasicLayout container

- absolute positioning and, 153
- coding example, 248
- constraints and, 173

behavior. *See also* skinning

behaviors

- defined, 285
- effects, 286–297
- filters, 306–312
- transitions, 297–306

BevelFilter effect, 287, 308

bin-debug folder, 28, 225, 259

binding. *See* data binding

BindingUtils class, 147

BitmapFill fill type, 325

blend modes, 287

Bloomquist, Tim, 120

BlurFilter effect, 287, 308

Boolean data type, 75

BorderContainer component

- addElement() method, 159
- functionality, 41, 153
- inline style assignment, 314
- nested layouts and, 154

border properties, 152

bottom property

- ButtonBar navigator, 248
- Button control, 273
- functionality, 173–174

Bounce effect class

- Accordion navigator and, 254–256
- easeIn method, 255
- easeOut method, 255

braces {}

- creating functions, 68
- <fx:binding/> tag and, 145, 146
- inline concatenation, 141, 142
- inline conditional and, 354
- one-way binding and, 138

brackets ([]), 286

breakpoints

- creating, 95
- Debug perspective, 96–101
- defined, 95
- placement recommendations, 95

browsers

- accessing applications, 6
- caching support, 263, 371
- navigation considerations, 370

ButtonBar control

- functionality, 210
- as navigator, 43
- selectedItem property, 252
- ViewStack and, 248

Button control

- click event, 282
- creating, 85
- defaultButton property, 233
- dragging into containers, 38
- functionality, 40
- inline ActionScript and, 65
- label property, 38, 85
- positioning properties, 272
- relative sizing and, 161
- resizing considerations, 160

C

C languages, 9

caching

- Flex applications, 371
- framework, 371
- SWF files, 263

Cairngorm Framework, 7

calculateAge() function, 202, 203

CallAction effect, 302

CamelCase naming convention, 64, 80, 264, 323

Cartesian coordinate system, 112

Cascading Style Sheets. *See* CSS (Cascading Style Sheets)

case sensitivity, 20, 57

- casting data types, 76
- CDATA tag, 67
- Chambers, Mike, 346
- change event
 - adding/removing children, 159
 - applying filters, 309
 - functionality, 104, 105
 - testing data input, 112
- changeState() function, 275
- character sequences, 143
- CheckBox control
 - change event, 159
 - ContractManager application, 353
 - functionality, 40
 - label property, 38
 - resizing considerations, 160
- children
 - accessing in layouts, 156–157
 - adding/removing, 158–159
 - descendant accessor and, 220
 - navigator support, 245
 - passing styles to, 141
 - rearranging, 160
- chromeColor style, 318
- chrome (graphics), 318
- class-based models, 149
- classes
 - adding to packages, 80
 - constructors and, 76, 81
 - creating, 80–81
 - custom, 83–84
 - data type assignment, 74
 - defined, 79
 - dot notation, 64
 - OOP terminology, 39
 - setting properties, 82
 - toString() method, 76
- click event
 - Button control, 282
 - calling functions, 69
 - functionality, 105
 - navigator support, 250
 - play() method, 286
 - triggering, 104
 - triggering even animations, 286
 - validate() method and, 193
- CLIP format (masks), 287
- code completion
 - CDATA block and, 68
 - dot notation and, 193
 - functionality, 55–56
 - inserting blank namespaces, 219
 - namespaces and, 61, 219
 - Object classes and, 79
 - ordering function parameters, 73
 - state declarations and, 277
- code editor, 32
- code hinting
 - functionality, 55
 - keyboard shortcut, 56, 193
 - tooggling, 333
- coercion, 76, 149
- ColdFusion application, 240
- Collision! Flash game
 - background information, 120–121
 - creating supporting classes, 121–124
 - customizing strategies, 134–136
 - functions in, 129–134
 - game play strategies, 134–136
 - initial insights, 127–128
 - main application code, 124–127
- colon (:), 74, 76, 318
- ColorMatrixFilter effect, 308
- color names, 318, 320
- ColorPicker component, 183, 191
- color property, 325
- color style, 317
- ComboBox control, 40, 210
- com.learningflex4.components
 - package, 329
- com.learningflex4.methods package, 329
- com.learningflex4.skins package, 329
- comma (,), 198
- command-line compiler, 16, 17, 431–440
- comments, 87, 317
- compiling code
 - checking options, 366–370
 - at command-line, 16, 17, 431–440
 - comments and, 88
 - errors of omission, 70
 - Flex 4 compiler API, 16
 - Linux considerations, 17
 - quotation marks and, 66
 - saving projects and, 216
 - semi-colon and, 65
 - XML data considerations, 222–224
- complete event, 104, 263
- components. *See also* specific types of components
 - adding in Source mode, 54–55
 - categories supported, 40
 - children, 38
 - coding in applications, 53
 - creating dynamically, 85
 - creating invitations for, 117
 - creating self-powered, 117–120
 - defined, 39
 - Display List and, 155–160
 - dragging/dropping, 36–38
 - Flex layout process and, 169
 - identifying events available, 105
 - including/excluding from states, 277–278
 - inheritance and, 109
 - linking properties to states, 278
 - naming conventions, 64
 - packages and, 40
 - “pay as you go” concept, 152
 - percentage-based sizing and, 161
 - selecting, 38
 - styleName property, 323
 - visible property, 244
- Components pane (Flash Builder)
 - depicted, 34
 - dragging/dropping components, 36–38
 - Layout controls, 37
 - nonvisual components and, 44
 - preparing forms, 178
- concatenation
 - building text strings, 73
 - data binding and, 140–142
 - defined, 66
- concatenation operator, 66, 83
- conditional operator, 354
- configuration files, 437–439
- Confirm Perspective Switch dialog, 95
- Connect to Web Service command, 240
- Console pane
 - displaying trace() statement, 90
 - ending debug session, 100
 - resizing display, 92
- constants, defined, 114
- constraints
 - code example, 157
 - common properties, 173–174
 - defined, 173

constructors

- creating, 76
- functionality, 76, 81
- passing values to, 76
- ContactManager application
 - background information, 347
 - code for, 349–356
 - configuration tasks, 348
 - creating AIR project, 347
 - deploying to the desktop, 376–386
 - linking to servers, 356–364
- contactsDGLItemSelect() function, 350
- containers. *See also* specific types of containers
 - absolute positioning, 153
 - adding/removing children, 158–159
 - advanced functionality, 165–168
 - binding properties, 176
 - chrome support, 318
 - closing, 54
 - commonly used, 41
 - defined, 41
 - Display List and, 155–160
 - <fx:binding/> tag placement, 146
 - hybrid, 154
 - id property, 156
 - label property, 43
 - layout styles supported, 152
 - navigation, 257
 - navigators and, 42
 - objects as, 78
 - “pay as you go” concept, 152
 - recent enhancements, 41
 - relative positioning, 153–154
 - sizing considerations, 160–162
 - whitespace between, 149
- context menus, 25
- controlBarContent class, 276
- controlBarLayout class, 276
- controls. *See also* specific types of controls
 - closing, 54
 - commonly used, 40
 - defined, 40
 - navigators and, 42
 - resizing based on properties, 160
- converting data types, 76
- ConvolutionFilter effect, 287, 308
- creationComplete event
 - Application container, 115
 - ContactManager application, 359
 - functionality, 105

- List control, 212
 - triggering even animations, 286
- creationPolicy property, 265
- CreditCardValidator, 190
- crossdomain.xml file, 228
- Crossfade effect, 287
- CSS (Cascading Style Sheets)
 - applying custom skins, 334
 - converting styles to, 327
 - external, 320–326
 - Flex support, 8, 315
 - namespaces and, 317
 - style selectors, 317–320
 - syntax overview, 317–320
- CSS inheritance, 317
- CSS selectors, 316
- curly braces. *See* braces { }
- CurrencyFormatter, 198
- CurrencyValidator, 190
- currentState property, 273, 277, 281
- currentTarget property, 94

D

- danqsyle, 376
- databases, creating, 414–419
- data binding
 - applying, 138–146
 - concatenation and, 140–142
 - creating dynamic layouts, 176
 - defined, 137
 - escape sequences, 142
 - filters and, 306
 - <fx:binding/> tag, 145–146
 - handling complex data, 147–149
 - jumping to source code for, 23
 - multilevel, 148–149
 - multiple destinations, 140
 - one-way, 138
 - photo gallery application and, 265–268
 - recursion and, 265
 - resizable application layouts, 143–144
 - timing mechanisms and, 150
 - two-way, 146, 265
 - variables, 138–139
 - when not appropriate, 150
 - whitespace and, 141
- Data Definition Language (DDL), 410
- dataField property, 215, 221
- DataGridColumn
 - dataField property, 221
 - width property, 216
- DataGrid control
 - advanced lists, 215–217
 - dragging/dropping support, 235
 - functionality, 210
 - handling XML data in, 221–222
- Data Manipulation Language (DML), 410
- data models
 - building, 147–148
 - defined, 147
 - multilevel bindings, 148–149
- dataProvider property
 - creating thumbnails, 264
 - Halo namespace and, 216
 - List control and, 212, 213
- Data Services pane (Flash Builder), 34
- data types
 - assigning, 74
 - casting, 76
 - converting, 76
 - defined, 75
 - fundamental types, 75, 86
 - implicit coercion, 76
 - strongly typed, 82
 - typing functions, 76–78
 - typing variables, 76
- data validation. *See* validator components
- Date class, 202
- Date data type, 240
- DateField control, 182, 184, 201
- DateValidator, 190
- DDL (Data Definition Language), 410
- Debug button, 90
- debugging
 - commenting code and, 88
 - ending sessions, 100
 - goal of, 89
 - inspecting properties via trace(), 93–94
 - multiple monitor setup, 93
 - outputting values via trace(), 90–93
 - using breakpoints, 95–101
- Debug mode
 - launching, 90, 95
 - process overview, 94
 - tools supporting, 97

- Debug pane (Debug perspective), 97–101
 - Debug perspective
 - Debug pane, 97–100
 - Variables pane, 96
 - Declarations block
 - HTTPService component and, 225
 - new projects and, 53
 - nonvisual components and, 53, 181, 197
 - transitions and, 298
 - Declarations tag, 53
 - default application, 139
 - defaultButton property, 233
 - DeHaan, Peter, 392
 - deploying applications
 - checking compiler options, 366–370
 - creating AIR installer, 376
 - customizing applications, 377
 - exporting installers, 378–382
 - exporting release build, 371–376
 - Flex framework library and, 370–371
 - seamless installations, 382–386
 - to the desktop, 376–386
 - to the Web, 365–376
 - descendant accessor, 220
 - descendant selectors
 - adding, 323
 - functionality, 318
 - order of precedence, 320
 - Design mode (Flash Builder)
 - adding components to applications, 36–39
 - adding style rules to CSS files, 327
 - applying custom skins, 334
 - beginning DataGrids, 215
 - binding expressions and, 143
 - building first project, 31–36
 - creating forms, 168
 - disabling snapping, 37
 - functionality, 22
 - identifying available events, 105
 - identifying available styles, 158
 - login/registration form, 275
 - managing view states, 271–275
 - modifying properties directly, 44–49
 - nonvisual components and, 44
 - preparing forms, 178
 - selecting components, 38
 - selecting navigator containers, 257
 - style/property differences, 315
 - toggling modes, 32, 51, 52
 - desktop applications
 - defined, 6
 - deploying, 376–386
 - destination property, 138
 - development environment
 - MAMP, 348, 395–398
 - PHP development tools, 398–407
 - WAMP, 348, 395–398
 - digital certificates, 378, 379
 - Digital Signature dialog, 378
 - DisplacementMapFilter effect, 308
 - Display List
 - accessing children, 156–157
 - default layer order, 155
 - defined, 155
 - Divided Box containers, 165–166
 - DML (Data Manipulation Language), 410
 - dollar sign (\$), 198
 - domain property, 189
 - dot (.), 319
 - dot notation
 - code completion and, 193
 - component properties and, 159
 - E4X support, 221
 - functionality, 64
 - styles and, 86
 - double forward slash (//) syntax, 87
 - double quotes (“”), 65, 66
 - draggable components
 - creating, 115–117
 - invitations to interact with, 117
 - list-based controls, 234–235
 - dragMoveEnabled property, 235
 - DropDownList control, 209
 - dropEnabled property, 234
 - DropShadowFilter effect, 287, 308
 - duration property, 293
- ## E
- E4X (ECMAScript for XML), 220–221
 - easeIn method, 255
 - easeOut method, 255
 - Eclipse
 - downloading, 14
 - Flash Builder plug-in, 16, 18, 19, 34
 - functionality, 18
 - integrating Flex 4 SDK, 14
 - XML Developer Tools, 398, 401–402
 - ECMAScript for XML (E4X), 220–221
 - editable property, 184
 - Editor pane (Flash Builder), 34
 - effects
 - applying Spark effects, 287–297
 - assigning targets, 286
 - basics overview, 286
 - categories supported, 286–287
 - explicit targeting, 295–297
 - in parallel, 290–293
 - passed-in targets, 295
 - in sequence, 290
 - skinning support, 335–343, 340
 - EmailValidator, 187, 192
 - embedded assets, 332
 - @Embed() directive, 332
 - empty string (“”), 68
 - enabled property, 48
 - encoding standards, 58
 - environment variables, 436
 - equality operator, 156, 275
 - equals sign (=), 66
 - error checking
 - external data, 223
 - FormItem container and, 185
 - errorString property, 191
 - escape sequences, 142, 197
 - event constants, 114
 - event handling
 - in ActionScript, 113–120
 - via inline ActionScript, 109–113
 - event listeners, 87, 109–120
 - event parameter, 93, 119
 - events
 - automating, 117
 - calling functions, 69
 - Collision! Flash game, 120–136
 - commonly used, 104–105
 - defined, 104
 - identifying available, 105, 106
 - inline ActionScript and, 65
 - inspecting properties via trace(), 93–94
 - listening for, 109–120
 - researching, 105–109
 - responding to, 109–120
 - triggering effect animations, 286
 - triggering functions, 90
 - exclamation mark (!), 283
 - excludeFrom property, 277, 278, 281
 - explicit sizing, 160

- exporting
 - installers, 378–382
 - release build, 371–376
 - setting export path, 374
- Export Release Build dialog, 374, 378, 380
- Extensible Markup Language. *See* XML (Extensible Markup Language)
- external CSS, 320–326
- external data
 - access considerations, 227
 - error checking, 223
 - loading at compile time, 222–224
 - loading at runtime, 224–228
- F**
- Facebook ActionScript API, 389
- Fade effect, 286, 287
- fadeOut effect, 286
- fault event, 227, 360
- faultHandler function, 227
- FDT editor, 14
- fields, adding, 416
- File menu (Flash Builder), 27
- File Transfer Protocol (FTP), 374
- fill class, 325
- Fill property, 201
- filter effects
 - applying, 308–312
 - functionality, 287, 306
 - target filtering, 298, 304–306
 - types of, 307–308
- filters property, 306, 309
- Fitchett, Michael, 391
- Flash Builder
 - adding comments, 88
 - alternative editors supported, 14–16
 - context menus, 25
 - creating projects, 27–28
 - downloading, 13
 - Eclipse plug-in, 16, 18, 19, 34
 - File menu, 27
 - free trial period, 7
 - functionality, 4
 - generating server-side code, 241
 - help system, 107
 - installing, 18–19
 - modes supported, 22
 - opening, 31
 - Profiler tool, 375
 - Project menu, 28
 - running applications, 20–27
 - as standalone installation, 16
 - workbench overview, 34
- FlashDevelop editor, 14, 15
- Flash perspective, 96
- Flash Platform
 - additional information, 4
 - background information, 5
 - elements of, 4
 - Language Reference, 107–109
- Flash Player
 - compiler options, 366
 - Flex support, 5, 8
 - functionality, 9
 - HTML wrapper options, 369, 370
 - installing, 431–433
- Flash XML Graphics. *See* FXG graphics
- Flex 3
 - arrays and lists in, 213
 - effect triggers, 297
- Flex 4
 - background information, 5
 - compiler API, 16
 - functionality, 1
 - Google public API, 231
 - identifying applications in, 11
 - language components, 1
 - open source, 7
 - reasons for using, 6–8
 - similarities to other technologies, 3, 8–10
 - Software Development Kit, 4
 - SWF file support, 3
 - when not to use, 11
- Flex 4 SDK
 - configuring FlashDevelop, 15
 - deploying applications, 366
 - downloading, 14, 434
 - integrating into Eclipse, 14
 - Linux and, 17
- Flex applications
 - adding components, 36–39
 - Application tag, 53
 - caching, 371
 - code completion, 55–56
 - compiling, 431–440
 - creating, 20
 - deploying to the desktop, 376–386
 - deploying to the Web, 365–376
 - designing accessible, 368
 - Google public API, 231
 - importing project archives, 20–22
 - MXML syntax rules, 56–60
 - navigating, 168
 - opening, 22–23
 - projects and, 20
 - running, 24
 - server considerations, 374
 - source code example, 52–54
 - whitespace considerations, 58
- FlexBeans plug-in, 17
- Flex Developer Center, 16
- Flex framework library, 370
- Flex layout process, 164, 169
- flexlib library, 389
- Flickr site, 264, 388
- FocusManager, 168
- folders
 - naming conventions, 27, 28
 - project, 435
- For..Each..In loop, 196, 352
- format() method, 198, 203
- formatString property, 184, 199
- formatters
 - combining restrictions and, 199–200
 - formatting input, 197–199
 - linking to functions, 200–206
- Form container
 - aligning components, 168
 - functionality, 42, 167–168
 - interpreting focus sequences, 168
 - starting forms, 178–179
- FormHeading control, 167, 178
- FormItem container
 - adding labels via, 230
 - aligning components, 168
 - error checking and, 185
 - functionality, 42, 167
 - label property, 196
 - preparing forms, 179
 - required property, 167
- forms
 - adding RadioButtonGroup, 180–181
 - combining restrictions/formatters, 199
 - completing layout, 182–184
 - controlling navigation, 254
 - creating, 168

- formatting input, 197–199
- linking formatters to functions, 200–206
- login/registration example, 275–279
- making inputs required, 179–180
- preparing, 178–179
- restricting input, 196–197
- validating data, 184–196
- Forta, Ben, 409, 410
- forward slash (/), 87
- framework caching, 371
- fromState property, 300, 304
- FTP (File Transfer Protocol), 374
- fully qualified references, 412
- function keyword, 68, 71
- functions
 - calling, 66, 69
 - controlling navigation via, 251–254
 - creating within Script/CDATA block, 68–70
 - defined, 66
 - getter, 357
 - linking formatters to, 200–206
 - methods as, 82
 - passing parameters, 72–74
 - passing variables, 68, 74
 - placement decisions, 67–68
 - return values, 76–78
 - scope and access level, 70–72, 72
 - setter, 357
 - triggering via events, 90
 - typing, 76–78
- <fx:binding/> tag
 - basic usage, 145
 - braces and, 145, 146
 - multiple sources and, 145–146
- FXG Editor, 6
- FXG graphics
 - adding to package directory, 335
 - applying Spark effects, 288–289
 - basic code block, 2
 - benefits, 2
 - changing color based on events, 117
 - Collision! game, 169
 - defining graphic borders, 325
 - fill graphic, 325
 - functionality, 2, 288
 - online editing, 6
 - Rect graphic, 201, 325
 - skinning support, 335–343
 - terminology, 315

G

- gaps, defined, 163–165
- garbage collection, 159
- gedit editor, 17
- getElementIndex() method, 156
- GET method (HTTP), 363
- getStyle() method, 86
- getter functions, 357
- global selectors
 - applying rules, 323
 - functionality, 318
 - order of precedence, 320
- GlowFilter effect, 287, 308, 338
- Google Maps API, 61, 389
- Google search engine, 231
- GradientBevelFilter effect, 308
- GradientEntry class, 325
- GradientGlowFilter effect, 308
- GraphicButtonSkin, 336–343
- Group container
 - functionality, 41, 153
 - nested layouts and, 154

H

- Haase, Chet, 392
- Halo namespace
 - dataProvider component and, 216
 - dataProvider property, 216
 - Flex 4 supported, 39, 60
 - List-based components and, 215
- Halo package
 - Divided Box containers, 165–166
 - effect support, 297
 - Form container, 167
 - FormItem container, 167
 - gap support, 163
 - handling XML data in, 221–222
 - layout styles supported, 152
 - List-based controls, 215
 - navigation components, 42, 43
 - padding layouts, 163
 - “pay as you go” concept, 152
 - sizing considerations, 160
- HBox container, 160
- headerText property, 215
- height property
 - AIR applications, 378
 - ButtonBar control, 248
 - HTML wrapper and, 369
- Image control, 261
- Hello World application
 - ActionScript interaction, 63, 64
 - using Design mode, 31–36
 - using Source mode, 52
- help system (Flash Builder), 107
- HGroup container
 - alignment considerations, 171
 - dragging into editor, 37
 - explicit sizing and, 160
 - functionality, 42
 - property considerations, 46
 - relative positioning, 153–154
 - selecting, 38
 - Spacer control and, 168
- horizontalAlign property
 - containers supporting, 154, 171
 - functionality, 172
 - Image control, 261
- horizontalCenter property, 175
- horizontalGap property, 163
- HorizontalLayout, 154, 171
- HorizontalList control, 210
- Host Component
 - assigning, 329–331
 - skinning support, 336
- Host Component selection dialog, 329
- HTML (Hypertext Markup Language)
 - AIR support, 346
 - background information, 5
 - Flex 4 similarities, 3, 9
- html-template folder, 28
- HTML wrapper, 369
- HTTP (Hypertext Transfer Protocol), 224, 363
- HTTPService class
 - connecting XML with, 260
 - ContactManager application, 356–364
 - functionality, 240
 - loading external data, 224–228
 - nonvisual components and, 44
 - photo gallery application, 266
 - sending data, 360–364
 - send() method, 224, 227
 - url property, 259
- hybrid containers, 154
- Hypertext Transfer Protocol (HTTP), 224, 363

I

id attribute (XML), 217

IDE (Integrated Development Environment), 9, 14, 18

id property

- accessing children, 156
- data binding and, 138
- fade effect, 286
- filters and, 307
- functionality, 46
- HTML wrapper and, 369
- preparing forms, 179
- ProgressBar control and, 40

id selectors

- creating, 324
- functionality, 319
- order of precedence, 320
- usage example, 334

If..Else block, 112, 134

Image control

- binding, 265
- complete event, 263
- functionality, 40
- height property, 261
- horizontalAlign property, 261
- open event, 263
- source property, 40, 261
- verticalAlign property, 261
- visible property, 263
- width property, 261

implicit coercion, 76

Import Existing Projects into Workspace dialog, 21

Import Flex Project Archive command, 21

importing

- existing projects, 26
- project archives, 20–22

import statement, 86, 118

includeInLayout property, 47, 244

includeIn property, 277, 278

index.html file, 375

inequality operator, 283

inheritance

- class, 111
- components and, 109
- CSS, 317

initialize event, 105

inline ActionScript

calling functions, 66, 69

controlling navigation, 250–251

functionality, 65

handling events, 109–113

passing parameters, 72

whitespace and, 141

inline conditional, 353, 354

inline styling

- inline style assignments, 314–315
- order of precedence, 320

insertContact.php script, 427

INSERT statement (MySQL), 413

int data type, 75

Integrated Development Environment (IDE), 9, 14, 18

IntelliSense, 55

interaction. *See* events

internal scope (functions), 71

invalidDomainError, 187

ItemRenderer class, 236–239

itemRenderer property, 236

J

Java EE (Java Enterprise Edition), 240

Java platform

- Flex 4 similarities, 9
- installing, 433

JavaScript

- AIR support, 346
- background information, 5
- Flex 4 similarities, 3, 9

K

KeyboardEvent class, 117, 119

keyboard shortcuts

- code hinting, 56, 193
- commenting in Flash Builder, 88
- debugging, 98, 100
- deleting projects, 25
- disabling snapping, 37
- Flash Builder Source mode, 23
- saving projects, 216
- toggle modes, 52

Key, Charlie, 391

keyDown event handler, 119

Knuth, Donald, 375

L

Label control

- binding to multiple sources, 145
- filters property, 309
- functionality, 40, 47
- Spacer control and, 168
- text property, 38, 47

labelField property, 214, 215

label property

- changing, 85
- control restrictions, 38
- FormItem component, 196
- functionality, 47
- layout containers and, 43
- NavigatorContent component, 245
- navigators and, 43
- preparing forms, 179
- removing from components, 168

Language namespace, 39, 60

Language Reference, 108–109, 158, 315

lastResult property, 227, 232

layout containers. *See* containers

left property

- Button control, 273
- functionality, 173–174

libs folder, 28

lightning bolt icon, 106

LinearGradient class, 325

LinkBar navigator

- functionality, 43, 249
- TabNavigator and, 249
- ViewStack and, 249

LinkButton component, 276

Linus' Law, 135

Linux operating system

- compiling applications, 431–440
- converting old systems to, 395
- Flex IDE and, 17
- installation file, 383

List-based controls

- custom item renderers and, 236–239
- dragging/dropping support, 234–235
- functionality, 209–210
- implementing list selection, 229–230
- lists of complex data, 213–217
- lists of simple data, 210–213

List control

- allowMultipleSelection property, 230
- creationComplete event, 212

- dataProvider property, 212, 213
- descendant selectors and, 323
- dragMoveEnabled property, 235
- dropEnabled property, 234
- functionality, 40, 209
- handling XML data, 217–228
- implementing list selection, 229–230
- labelField property, 214, 215
- lists of complex data, 213, 214
- lists of simple data, 211, 212
- selectedIndex property, 251
- selectedItem property, 229, 230
- loadContacts.php script, 425–427
- local scope, 72
- logical AND operator (&&), 309
- logical OR operator (||), 310
- login/registration form, 275–279
- Lott, Joseph, 308, 346, 388
- lowerCamelCase naming convention, 64, 323
- LUMINOSITY format (masks), 287

M

- Mac operating system
 - installation file, 383
 - MAMP, 348, 395–398
- MAMP (Mac OS), 348, 395–398
- masking
 - formats supported, 287
 - input fields, 197
- Math class
 - cos function, 130
 - floor() method, 203
 - PI function, 130
 - sin function, 130
- maximum sizes, 162
- MenuBar navigator, 43
- message property, 195
- methods. *See also* specific methods
 - classes and, 79, 82
 - defined, 82
- minHeight property, 162
- minimum sizes, 162
- minLength property, 186
- minWidth property, 162
- Module component, 371
- mouseDown event, 105
- MouseEvent class, 111, 116
- mouseLeave event, 114
- mouseOver event
 - class inheritance and, 111
 - triggering even animations, 286
- mouseUp event, 105
- Move3D effect, 287
- Move effect
 - functionality, 287
 - search application example, 301
- Mustafa, Nishad, 391
- MXML
 - basic code block, 2
 - commenting code, 87
 - compiling applications, 436
 - controlling navigation, 250
 - creating ArrayCollection, 210
 - functionality, 1, 56–60
 - <fx:binding/> tag, 145–146
 - identifying available styles, 158
 - inline ActionScript and, 65
 - login/registration form, 277
 - namespace designations, 60
 - nested properties and, 61
 - relationship with ActionScript, 85–87
 - Style tag, 316
 - viewing ActionScript code, 85
 - XML syntax rules, 56–60
- MySQL
 - creating databases, 414–419
 - INSERT/UPDATE queries, 360, 413
 - language elements/syntax, 410–411
 - overview, 345–346, 409
 - PHP functions, 423–424
 - SELECT queries, 412
 - SQL, DML, and DDL, 410
 - statements supported, 411–414
- mysql_connect() function, 423
- mysql_fetch_array() function, 424
- mysql_query() function, 423
- mysql_select_db() function, 423

N

- \n (new line), 83
- name property, 369
- name selectors
 - functionality, 319
 - order of precedence, 320
- namespaces
 - assigning, 53
 - code completion and, 61, 219
 - in CSS, 317
 - defined, 39, 60
 - Flex 4 supported, 39, 60–62
 - packages and, 60
 - third-party libraries and, 61
- naming conventions
 - CamelCase, 64, 80, 264, 323
 - for components, 64
 - for constants, 114
 - for folders, 27, 28
 - for packages, 62, 79, 329
 - reverse domain package naming, 62
 - style rules, 323
 - using abbreviations, 64
- National Oceanic and Atmospheric Administration (NOAA), 240
- navigateToURL() function, 239
- navigation
 - browsers and, 370
 - controlling via inline ActionScript, 250–251
 - Tab key, 168
- navigator components. *See also* specific types of navigators
 - click event and, 250
 - commonly used, 42–43, 244–256
 - creationPolicy property, 265
 - defined, 42, 244
 - label property and, 43
 - selecting containers, 257
 - view states versus, 271
- NavigatorContent component, 245
- Neil, Theresa, 111, 392
- nested content
 - application layouts and, 154
 - effects and, 294, 301
 - gaps in whitespace, 163–165
 - MenuBar and, 43
 - namespaces and, 61
 - navigators handlings, 43
 - XML tags, 57–60
- NetBeans IDE, 17
- Network Monitor pane (Flash Builder), 34
- New Flex Project dialog, 32
- new keyword, 86, 159
- New MXML Item Renderer dialog, 264
- New MXML Skin dialog, 330
- New State button, 271, 272
- New Style Rule dialog, 327

NOAA (National Oceanic and Atmospheric Administration), 240

Noble, Joshua, 287

nonvisual components

adding in Source mode, 260

Declarations block and, 53, 181, 197

formatters as, 197

RadioButtonGroups and, 181

validators as, 184

Notepad++ editor, 14

Number data type, 75, 240

NumberFormatter, 202

NumberValidator, 190

O

object-oriented programming (OOP), 39, 78

objects

classes and, 79

creating, 79

defined, 78

functionality, 78

OOP terminology, 39, 78

setting properties, 79

onAppComplete() function, 226

onEdit() function, 351, 361

one-way binding, 138

onFault() function, 361

online resources, 391

onNew() function, 351

OOP (object-oriented programming), 39, 78

open event, 263

OpenLaszlo, 10

operators

addition, 66, 142

additional information, 66

assignment, 66, 83, 196

concatenation, 66, 83

conditional, 354

equality, 156, 275

inequality, 283

logical AND, 309

logical OR, 310

ternary, 354

type, 74

Outline pane (Flash Builder)

depicted, 34

functionality, 19, 38

selecting components, 38

visualizing application structure, 171

P

Package Explorer pane (Flash Builder)

creating applications in projects, 139

deleting projects, 25

depicted, 34

functionality, 19

linking packages, 80

viewing packages, 329

viewing projects, 21

packages

classes and, 79–80

components and, 40

creating custom skins, 328–329

defined, 39, 79

expanding directory, 335

FXG graphics support, 335

linking to projects, 80

namespaces and, 60

naming conflicts, 62

naming conventions, 62, 79, 329

reverse domain package naming, 62

padding layouts, 162

paddingLeft property, 149

Panel container

chrome support, 318

feeding formatted output to, 200–206

functionality, 42

label property, 38

list selection example, 229

nested layouts and, 154

title property, 246

panels

default docking location, 34

defined, 19, 34

panes

accessing properties quickly, 45

closing, 35

defined, 19, 34

reopening, 35, 38

undocking, 34

Parallel block

effects in parallel, 290–293

search application example, 301

Sequence block and, 293–294

target filtering and, 305

parameters

event, 93

function, 72–75

parentheses (), 68, 72, 76

PDT (PHP Development Tools), 398–407

percentage-based sizing, 161

performance assessment, 375

perspectives

defined, 19, 96

remembering custom arrangements, 35

PhoneFormatter, 199

PhoneNumberValidator, 188

photo gallery application

adding tab views, 256–258

creating, 256

creating thumbnails, 264

CSS file and, 322–324

deploying to the Web, 365–376

displaying external images, 261

monitoring image loading progress, 262–263

populating with XML, 258–261

synchronizing lists, 265–268

PHP Development Tools (PDT), 398–407

phpMyAdmin, 414–419

adding fields, 416

browsing data, 417–419

creating databases, 414

creating tables, 414

inserting records, 417

PHP scripting language

ContactManager application, 358

external data services and, 241

language elements/syntax, 421–424

overview, 345–346

sample scripts, 424–428

pipe (|), 43, 318, 319

pixel-shading effects, 287

play() method, 286, 295

PNG graphic, downloading, 332

POST method (HTTP), 363

pound symbol (#), 199

primary key field, 414

print resources, 390

private scope (functions), 71

Problems pane (Flash Builder)

depicted, 34

functionality, 19

Profiler tool, 375

ProgressBar control

functionality, 40

monitoring image loading, 262–263

progress events and, 104

progress event, 104

- project folders, 435
 - Project menu (Flash Builder)
 - Properties option, 28
 - projects
 - changing settings, 28
 - creating, 27–28, 31–36, 347
 - deleting, 25
 - Flash Builder recall when opening, 52
 - Flex applications and, 20
 - folder considerations, 27, 28
 - importing, 26
 - importing archives, 20–22
 - linking to packages, 80
 - multiple application support, 139
 - naming considerations, 27, 32
 - saving, 216
 - setting location, 32
 - tweaking configuration files, 437–439
 - properties
 - accessing via Properties pane, 44–45
 - adding state-specific, 281–284
 - assignment and concatenation, 66
 - attributes as, 85
 - background, 152
 - border, 152
 - class, 79, 82
 - commonly used, 46–49
 - constraint, 173–174
 - controls resizing based on, 160
 - defined, 315
 - dot notation, 64
 - editing for view states, 272
 - inspecting via trace(), 93–94
 - linking components to states, 278
 - object, 79
 - strongly typed, 82
 - style, 86, 158
 - Properties pane (Flash Builder)
 - Alphabetical view, 36, 45
 - Category view, 36, 45
 - depicted, 34
 - identifying available events, 105
 - identifying available styles, 158
 - modifying properties directly, 44–49
 - preparing forms, 178
 - Source mode support, 44
 - Standard view, 36, 44
 - tweaking settings, 35
 - property attribute, 185
 - property effects, 287, 290
 - protected scope (functions), 71
 - public scope (functions), 71
- ## Q
- query property, 231
 - question mark (?), 330
 - quotation marks
 - attribute definition and, 65
 - click event and, 66
 - whitespace inside, 141
- ## R
- \r (carriage return), 83
 - RadialGradient class, 325
 - RadioButton control
 - functionality, 41
 - label property, 38
 - RadioButtonGroup component
 - adding to forms, 180–181
 - selectedValue property, 201
 - Raymond, Eric S., 135
 - records, inserting, 417
 - recursion, binding and, 265
 - reference materials, 392
 - refreshApp() function, 352, 359
 - RegExpValidator, 190
 - registration form, 275–279
 - regular expressions, 190
 - relative positioning, 153–154
 - relative sizing, 161
 - release build
 - AIR applications, 378
 - exporting, 371–376
 - uploading to web host, 374
 - RemoteObject component, 240
 - remote objects, defined, 240
 - RemoveAction effect, 302
 - removeElement() method, 158, 159
 - renderText() function, 238, 239
 - Republic Of Code website, 135
 - requiredFieldError, 186
 - required property
 - FormItem container, 167
 - preparing forms, 179, 180
 - Resize effect
 - functionality, 287
 - search application example, 301
 - resize event
 - functionality, 105
 - triggering even animations, 286
 - resources
 - online, 391
 - print, 390
 - reference materials, 392
 - restrict property, 197
 - resultFormat property, 226
 - Resume command, 98
 - return values, functions and, 76–78
 - reverse domain package naming, 62
 - RIA (Rich Internet Application)
 - additional information, 314
 - creating FXG graphics code, 288
 - defined, xi
 - right property
 - Button control, 273
 - functionality, 173–174
 - Robison, Steve, 120
 - rollOut event, 105
 - rollOver event, 105, 111
 - root tag
 - application requirements, 53, 148
 - for components, 122
 - defined, 217
 - XML requirements, 217, 218
 - Rotate3D effect
 - effects in parallel, 290
 - functionality, 287
 - search application example, 301
 - skinning example, 340
 - Rotate effect, 287
 - rotation property, 325
 - rounding property, 203
 - RSL (Runtime Shared Library), 370
 - RSS feeds, 388
 - Run button (toolbar), 24
 - runtime, loading external data at, 224–228
 - Runtime Shared Library (RSL), 370
- ## S
- Sabbath, Matthew, 431–440
 - sandbox, defined, 228
 - Scale3D effect, 287
 - Scale effect, 287
 - Schinsky, Holly, 238, 391
 - scope and access level
 - for functions, 70–72, 72
 - for variables, 74
 - Scott, Bill, 111

- Script/CDATA block
 - accessing functions, 70–72
 - creating, 67
 - creating functions within, 68–70
 - data types, 75–78
 - function parameters, 72–74
 - function placement, 67–68
 - variables and, 74
- scripting
 - Bash scripts, 439
 - changing current state, 273–275, 279
 - inline ActionScript, 65
 - Script/CDATA block, 67–74
- SDK (Software Development Kit), 13
- search application
 - connecting to results, 231–234
 - effect examples, 299–306
 - restoring, 279
 - revising, 280–281
- SearchService component, 231, 232
- security
 - digital certificates, 378, 379
 - sandbox considerations, 228
- selectedChild property, 250, 251
- selectedColor property, 191
- selectedDate property, 184, 201
- selectedIndex property, 40, 250, 251
- selectedItem property
 - ButtonBar navigator, 252
 - ComboBox control, 40
 - List control, 229, 230
- selected property, 159
- selectedValue property, 201
- selectors, defined, 317
- SELECT statement (MySQL), 412
- semicolon (;), 65, 74, 318
- send() method
 - HTTPService component, 224, 227
 - search application example, 282
 - SearchService component, 231
- Sequence block
 - nesting sequenced effects, 290
 - order of effects, 290, 302
 - Parallel block and, 293–294
- setAction effect, 302
- setElementIndex() method, 156
- setStyle() method, 86, 158, 159
- setter functions, 357
- 7Jigen Labs, 6, 288
- Shacklette, Justin, 116
- shorthand conditional, 354
- show() method, 193, 196
- Silverlight, 10
- single quotes (')
 - in attribute definitions, 65
 - in binding expressions, 143
 - click event and, 66
 - in escape sequences, 142
 - inline concatenation and, 141
- sizing
 - explicit, 160
 - minimum/maximum, 162
 - percentage-based, 161
 - relative, 161
- SkinnableContainer, 153, 154
- skinning
 - audiotool example, 314
 - creating custom skins, 329–334
 - defined, 3, 313, 326
 - effects support, 335–343
 - FXG graphics support, 335–343
 - working with packages, 328–329
- snapping indicators
 - disabling, 37
 - influencing drops, 37
- SOAP web services, 240
- SocialSecurityValidator, 190
- Software Development Kit (SDK), 13
- SolidColor fill type, 325
- SolidColor property, 201
- source code, sharing, 372–373
- source control systems, 18
- Source mode (Flash Builder)
 - adding components, 54–55
 - adding nonvisual components, 260
 - applying custom skin, 332–333
 - changing current state, 273–275
 - cleaning up code, 53
 - code completion, 55–56
 - Declarations block, 53
 - finetuning component positioning, 37
 - functionality, 22
 - identifying available events, 106
 - identifying available styles, 158
 - keyboard shortcut, 23
 - login/registration form, 275, 277
 - MXML support, 56–60
 - namespaces and, 60–62
 - nonvisual components and, 44, 184
 - preparing forms, 178, 179
 - Properties pane support, 44
 - source code example, 52–54
 - style/property differences, 315
 - toggle modes, 32, 51, 52
- source property
 - Image control, 40, 261
 - MXML Style tag, 316
 - namespaces and, 317
 - pointing to external data, 222
 - Style tag, 322
 - validator components, 185
- Spacer control, 168–169
- Spark namespace
 - Flex 4 supported, 39, 60
 - List-based controls, 215
- Spark package
 - absolute positioning, 153
 - effect support, 297
 - filter support, 307
 - gap support, 163
 - layout styles supported, 152
 - navigation components, 42, 43
 - NavigatorContent component and, 245
 - padding layouts, 163
 - “pay as you go” concept, 152
 - sizing considerations, 160
- special characters, 143, 197
- SQLite database, 346
- src folder, 28, 225
- Stage class, 119
- stageHeight property, 119
- stageWidth property, 119
- states. *See* view states
- State selection box, 272
- States pane (Flash Builder)
 - creating new state, 271
 - depicted, 34, 271
- Step Into command, 98
- Step Over command, 98
- Step Return command, 98
- strict type checking, 367
- String data type, 75
- StringValidator, 185–187
- stroke class, 325
- strongly typed data, 82, 357
- style attributes, 86, 315

Style block
 creating, 322
 declaring, 316
 namespace definitions, 317
 opening, 317
 styleName property, 323
 styles
 adding rules to CSS file, 322–324
 converting to CSS, 327
 defined, 3, 86, 313
 inline assignments, 314–315
 order of precedence, 320
 passing to children, 141
 properties versus, 315
 property values of, 86, 158
 rule syntax, 318
 setting with attributes, 86
 Subclipse plug-in, 18
 submitComplete() function, 352
 “sun angle” effect, 325
 SVN (Subversion) repository, 18
 SWC files, 231
 SWF files
 caching, 263
 deployment considerations, 367
 Flash support, 9
 Flex 4 support, 3
 size considerations, 224
 Switch..Case block, 127, 134, 252
 Swiz Framework, 7

T

TabBar navigator
 functionality, 43
 ViewStack and, 246–248
 tab index, 368
 tabIndex property, 168
 Tab key, 168, 185, 368
 tables, creating, 414
 TabNavigator
 coding example, 245, 246
 creationPolicy property, 266
 functionality, 42, 244–246
 LinkBar and, 249
 name selectors and, 323
 photo gallery application, 256
 selecting containers, 257
 tags
 adding attributes to, 85
 anatomy of, 58–60

angle brackets and, 56, 67
 creating, 57
 creating relationships, 56
 nested, 57
 syntax rules, 56, 57
 target filtering, 298, 304–306
 target property, 296
 Terminate button, 100
 ternary operator (inline If...Else), 354
 testing
 applications, 304, 376
 installations, 398
 PHP Development Tools, 403–407
 XML Developer Tools, 401–402
 TextArea control
 code example, 83
 functionality, 41
 item renderers and, 238
 preparing forms, 182
 textFlow property, 238
 text property, 38
 Text control, 41
 TextFlow class, 239
 textFlow property, 238
 TextFlowUtil class, 239
 TextInput control
 adding to forms, 178
 dragging into container, 38
 functionality, 41
 preparing forms, 182
 query property, 231
 restricting input, 196
 text property, 38, 64
 TextMate editor, 14
 TextPad editor, 14
 text property
 binding considerations, 138
 binding to multiple controls, 146
 control considerations, 38, 47
 dot notation example, 64
 functionality, 48
 namespace designations and, 61
 Text control and, 41
 TextInput control and, 41
 themes
 defined, 343
 selecting, 3

3D effects, 287
 thumbList control, 264, 265
 thumbnails, creating, 264
 TileGroup container, 167
 TileLayout container, 167
 TileList control, 235
 time property, 202
 Timer class, 117, 119
 title property, 246, 350, 369
 toolTip property
 creating line break, 264
 disabled components and, 48
 functionality, 47
 tooShortError property, 186
 top property, 173–174
 toState property, 300, 304
 toString() method, 76
 trace() statement
 determining height via, 248
 displaying, 90
 inspecting properties, 93–94
 outputting values via, 90–93
 transform effects, 287
 transitions
 adding animation/effects, 340
 building, 300–304
 defined, 286
 order of effects, 302
 syntax overview, 298–299
 target filtering and, 298, 304–306
 triggerEvent property, 192
 trigger property, 192
 trigonometric functions, 130
 Tucker, David, 346
 tweener library, 389
 Twitter ActionScript/Flash APIs, 389
 twoWay attribute, 147
 two-way binding, 146, 265
 type declarations
 error example, 71
 type operator, 74
 type property, 94
 type selectors
 applying, 323
 functionality, 319
 order of precedence, 320

U

- Ubuntu, 17, 431–440
- UIComponent base class, 46, 111
- uint data type, 75
- Umap API, 389
- underscore (`_`), 114
- updateContact.php script, 428
- updateFilters() method, 309
- UPDATE statement (MySQL), 413
- UpperCamelCase naming convention, 64, 80, 264
- url property, 259
- URLRequest class, 239, 241

V

- validateAll() method, 193–196
- validateAndSubmit() function, 193
- validate() function, 352
- validate() method, 193
- ValidationResultEvent class, 195
- Validator class, 193–196
- validator components
 - custom techniques, 192–196
 - functionality, 184, 185–191
 - source property and, 185
- variables
 - assigning return values to, 77
 - binding, 138–139
 - creating, 74
 - data type assignment, 74
 - declaring, 71, 74
 - defined, 74
 - passing into functions, 68
 - setting default values, 74
 - setting scope, 74
 - typing, 76
- Variables pane (Debug perspective)
 - functionality, 96
 - location, 96
- var keyword, 74
- version numbers, 369
- verticalAlign property
 - containers supporting, 171
 - functionality, 172
 - Image control, 261
- verticalCenter property, 175
- verticalGap property, 163
- VerticalLayout, 154, 171
- verticalScrollPolicy property, 238
- VGroup container
 - alignment considerations, 171
 - explicit sizing and, 160
 - functionality, 42
 - property considerations, 46
 - relative positioning, 153–154
 - search application example, 282
- ViewStack navigator
 - ButtonBar and, 248
 - controlling with ActionScript, 250–254
 - functionality, 43, 246
 - LinkBar and, 249
 - selectedChild property, 250, 251
 - selectedIndex property, 250
 - TabBar and, 246–248
- view states
 - adding, 281–284
 - applying to search application, 279–284
 - changing via scripting, 273–275, 279
 - creating, 271
 - editing properties, 272
 - effects and, 299–306
 - including/excluding components, 277–278
 - linking component properties, 278
 - login/registration form example, 275–279
 - managing in Design mode, 271–275
 - modifying layouts, 272–273
 - navigators versus, 271
 - scenarios for, 269–271
 - transitions and, 298
- visible property
 - functionality, 46, 244
 - Image control, 263
- void data type, 75, 77

W

- WAMP (Windows), 348, 395–398
- Web 2.0, xi
- web applications
 - defined, 6
 - deploying, 365–376
- WebSearchResult class, 238
- WebService component, 240

- Web Services Description Language (WSDL), 240
- whitespace, 58, 141
 - controlling in layouts, 162–165
 - padding layouts, 162
 - paddingLeft property and, 149
- width property
 - AIR applications, 378
 - DataGridColumn, 216
 - Image control, 261
 - ItemRenderer class, 237
- WindowedApplication container, 346, 350
- WindowedApplication tag, 53
- Window menu (Flash Builder), 35, 38
- Windows operating system
 - installation file, 383
 - WAMP, 348, 395–398
- Wipe effect, 287
- workspaces
 - copying, 26
 - creating, 26
 - default, 26, 52
 - defined, 26
 - Flash Builder recall when opening, 52
- WSDL (Web Services Description Language), 240

X

- XAML, 10
- XML data
 - handling as XMLListCollection, 219
 - handling in Halo DataGrid, 221–222
 - loading at compile time, 222–224
 - loading at runtime, 224–228
 - parsing, 219
 - reading using E4X, 220–221
 - structure overview, 217–219
- XML Developer Tools (Eclipse), 398, 401–402
- XML (Extensible Markup Language)
 - CDATA tag, 67
 - Flex support, 8
 - functionality, 56

- id attribute, 217
- populating photo gallery, 258–261
- syntax rules, 56–60
- whitespace considerations, 58
- xml folder, 225
- XML() function, 359
- XMLListCollection class, 210, 219
- xmlns="" attribute, 219
- x property, 46

Y

- Yahoo! Maps API, 389
- yahoo namespace, 231
- YahooSearch application. *See* search application
- yearNavigationEnabled property, 182
- YouTube API, 389
- y property, 46

Z

- Z-axis, 287
- ZipCodeValidator, 188
- ZipCodeValidatorDomainType class, 189

