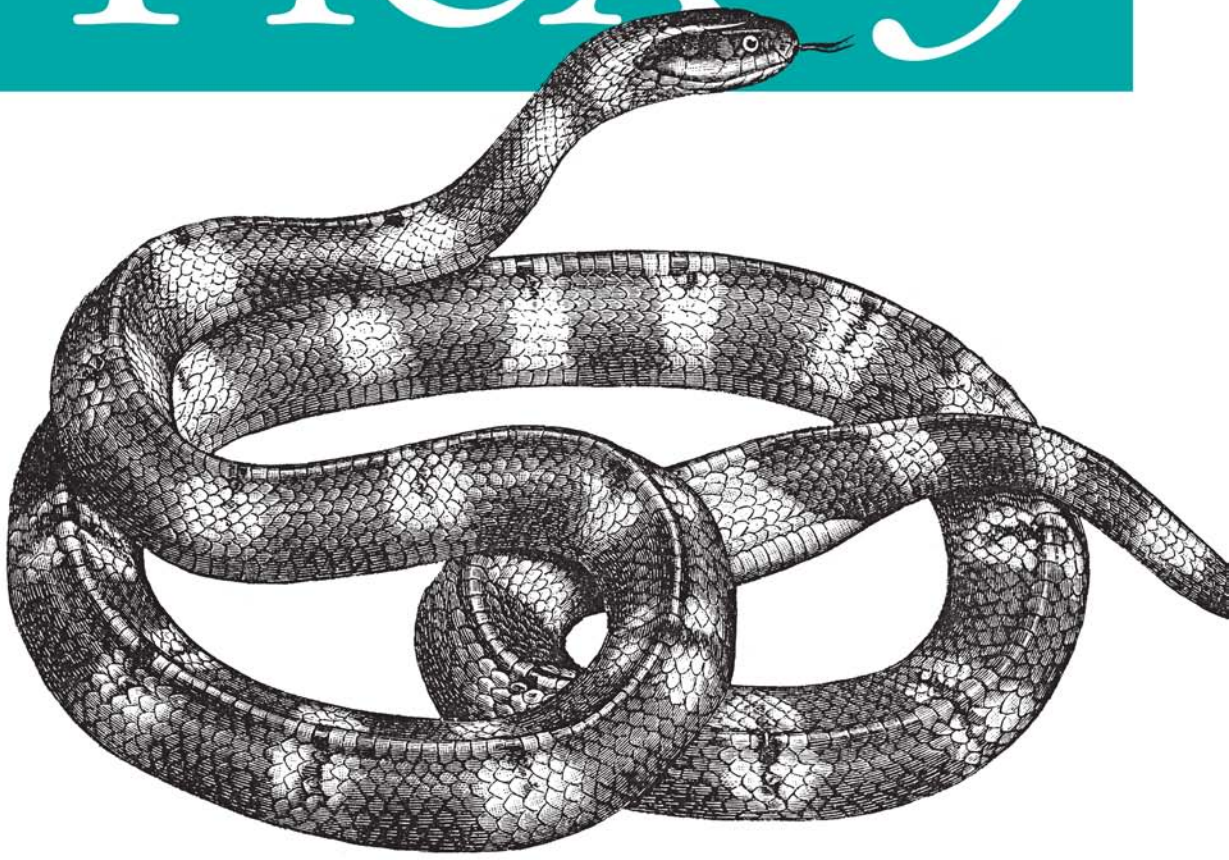
The Comprehensive Guide to Creating Rich Internet Applications with Adobe® Flex™

Programming

Flex™ 3



O'REILLY®

Adobe Developer Library

Chafic Kazoun & Joey Lott

O'REILLY®

Programming Flex 3

If you want to try your hand at developing Rich Internet Applications with Adobe's Flex 3, this is the ideal book to get you started. *Programming Flex 3* gives you a solid understanding of Flex 3's core concepts, and valuable insight into how, why, and when to use specific Flex features. Numerous examples and sample code demonstrate ways to build complete, functional applications for the Web using the free Flex SDK, and ways to build RIAs for the desktop using Adobe AIR. This book is an excellent companion to Adobe's Flex 3 reference documentation. With this book, you will learn about:

- The underlying details of the Flex framework
- Programming with MXML and ActionScript
- Architecture and layout of UI components
- Best practices for working with media
- Management of states for applications and components
- Tips for using transitions and effects
- Debugging Flex applications
- Embedding Flex applications in web browsers
- Building AIR applications for the desktop

Flex 3 will put you at the forefront of the RIA revolution on both the Web and the desktop. *Programming Flex 3* will help you get the most from this complex and powerful technology.

"Chafic and Joey are true Flex experts and provide an excellent resource for developers, new or not."

—Matt Chotin, Senior Product Manager, Flex Adobe Systems, Inc.

Chafic Kazoun is co-founder and Chief Software Architect at Atellis. He's worked with Flash technologies since 1998 and with Flex since its inception.

Joey Lott is a founding partner in The Morphic Group (*www.themorphicgroup.com*), specializing in Flex application development. Joey has also written many other leading books on Flex and Flash-related technologies, including O'Reilly's *ActionScript 3.0 Cookbook*.

www.oreilly.com
www.adobedeveloperlibrary.com

US $54.99 **CAN $54.99**

ISBN: 978-0-596-51621-5

Programming Flex™ 3

Chafic Kazoun and Joey Lott

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Programming Flex™ 3
by Chafic Kazoun and Joey Lott

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Audrey Doyle
Production Editor: Michele Filshie
Proofreader: Kim Wimpsett

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:
September 2008: First Edition.

This book uses RepKover, a durable and flexible lay-flat binding.

ISBN: 978-0-596-51621-5

[C]

1220557168

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 *adobedeveloperlibrary.com*.

Table of Contents

Foreword

I remember 2004. That was the year the Olympics were held in Greece. Oil rose above $50 per barrel. *The Return of the King* swept the Oscars. The Red Sox won the World Series. The Serendib Scops Owl was discovered in Sri Lanka... What a year! Okay, that last one I ripped off from Wikipedia. But 2004 *was* a big year for those owls... and it was also a big year for Internet applications. It was the year Flex was born.

A lot has changed in just a few short years. Flex 1 was very exclusive and its applications were tied to a server. It required expensive licenses and few resources were available to help you out. Flex 1.5 cut the cord between the application and the server. Suddenly, anyone could write and deploy a killer Flex app, but most folks still had not heard of Flex. When Flex 2 came out, it was really making some headway into the mindshare of rich Internet application (RIA) developers, even as the industry struggled to define what a RIA developer was. Flex got more and more press, and the SDK was finally released for free. By the time the 2.0.1 update shipped, Flex had an impressive following of designers, developers, so-called *devigners*, and that rarest of beasts, the Serendib developer.

And now comes Flex 3, the most complete and usable version of Flex yet. You get a profiler, OLAP, CS3 integration, refactoring, framework RSLs, deep linking, an AJAX bridge, code generation for servers, automation, just about everything you could dream of. And if something isn't in the box, you can bet someone in the community is working on it: frameworks, 3D libraries, maps, mashups, configurators, dashboards, monitors, widgets, you name it.

But with all those new features and functionality, what's the biggest change in Flex 3?

Well, it's not a new feature, or a refactored API. It's not the splashy new box cover, and it's not the low, low price. It's not even that snazzy new "Getting Started Experience." No, it's none of these things. To see the biggest change in Flex 3, to really see it, you need to stand up, walk down the hall, step into the bathroom (after knocking politely, of course), and look in the mirror. The biggest change in Flex 3 is *you*. That's right. With Flex 3, you, I, or anyone else can contribute to the open source Flex SDK. You can stick your hand into the belly of the beast, tweak its spleen, sew it up, and

reawaken a whole new beast. With just a text editor and an Internet connection, you can become a contributor on this leading RIA technology.

So, where does this book fit in? Looking at the existing Flex 3 product documentation, I see more than 2,300 pages of content and nearly 1,200 example applications. I even wrote a couple of those, although if you corner me with a compiler error, I'll deny it. And that doesn't even include the Language Reference, with thousands more "virtual" pages of developer doc. So, why do we need a book about Flex 3 if so much content is already available?

Well, when they wrote *Programming Flex 2*, the first edition of this book, Chafic and Joey learned how to use Flex 2 from the outside in. This was before the source code was even available to look at. They managed to figure out how to do such things as work with remote data, navigate the complexities of the Flex layout schemes, and create incredible custom components. They were real developers solving real problems and writing real code. I remember looking at many of the topics in that edition and saying to myself, "I wish I had written that." These guys took incredibly complex topics and distilled them into the information you needed.

For this edition, Chafic and Joey looked at the product from the inside out. They peeled back the skin and saw the sinewy skeleton of a dynamic framework that will define the next generation of web apps. If you're designing a video player, there's a chapter for you. If you've got a yen for currency formatters, this book has you covered. If you just want to get a handle on the application life cycle, you came to the right place.

So, this book will tell you what Flex 3 is. And after you read it, you might discover something that Flex 3 isn't. But now there's something you can do about it. At some late hour, when everyone else is asleep, if the inspiration strikes you, you might screw up your courage and heap on the moxie, and put your mark on the Flex world by joining the forces at *http://opensource.adobe.com/flex*. This book is just the beginning.

—Matt Horn
Adobe

Preface

It literally took us several years to write *Programming Flex 2*, the predecessor to this book. We worked hard on that book, and when it was finally written and edited and proofread and off to the printer we sighed and looked forward to a break from writing about Flex. However, Flex 3 followed close on the heels of Flex 2, and as the saying goes, there's no rest for the weary. We again picked up our keyboards and started updating the book for Flex 3. The result is what you have in your hands. And it is more than a simple update.

We thought *Programming Flex 2* was one of the best books available for Flex 2. However, we knew we could do better. There were topics we just didn't have time to include in that book. With *Programming Flex 3* we wanted to not only update the book for Flex 3, but also expand our coverage to include things that weren't in the first book. We think we achieved that goal.

The most notable additions to *Programming Flex 3* are in Chapter 20, Chapter 21, and Chapter 22. In Chapter 20, we go into great detail on everything you need to know to add Flex applications to web pages, which we think is an important (if not crucial) topic. Chapter 21 covers building Adobe AIR desktop applications using Flex. And Chapter 22 contains the synthesis of everything else we discuss throughout the book. This is the one addition we think is perhaps the most important, since it helps explain how to take everything you've learned about Flex in preceding chapters and use that knowledge to build a real-world application.

However, we didn't merely add new chapters to the book. We also revised and updated all the chapters in the book. Some chapters didn't require much updating because there were minimal changes for the relevant features between Flex 2 and Flex 3. On the other hand, other chapters required extensive updates and additions. If you read Programming Flex 2 then you'll find lots of new or revised content in this book.

Flex 3 is huge in scope, even bigger than Flex 2. Although the learning curve is not steep (it's actually very easy to get started building Flex 3 applications), it is a long learning curve simply because of the massive amount of features packed into the framework. The official Flex documentation is quite good at telling you how to do something once you know what you're looking for. Therefore, we made it our goal to present to you a book that fills in the gaps and helps you to get comfortable enough with Flex that you

can start using it right away. It is our intention in this book to provide you with practical advice from our own experiences learning Flex, and from our longer-term experiences building rich Internet applications using Flash Platform technologies.

We really feel that Flex 3 is a fantastic product and a great way to build applications. Although this is a technical book, we have poured our enthusiasm into our writing, and we'd like to think you will share our enthusiasm as you read this book. We feel that Flex 3 is a far better way to build rich Internet applications than any alternative currently on the market, and we think that as you read this book and learn how to work with Flex, you'll agree. With Flex, you have few (if any) problems involving cross-browser compatibility, network data communication is a snap, and the framework is built with solid object-oriented principles and standards in mind. In short, we feel it's the fastest way to build the coolest, most stable applications.

Who This Book Is For

This book is intended for anyone looking to learn more about Flex 3. We recognize that the audience for this book represents a very diverse group of people with many different backgrounds. Some readers may already be experts at working with Flex 2 (though they may be new to Flex 3), whereas others may never have heard of Flex before picking up this book. Some readers may have years of experience working with Flash Platform technologies, and others may be completely new to creating content that runs in Flash Player. Some readers may have computer science degrees or may have worked in the software industry for years. Yet others may be self-taught. We have done our best to write a book that will cater to this diverse group.

However, be aware that to get the most from this book, it is best that you have a solid understanding of object-oriented principles and that you are comfortable with understanding concepts such as runtime environments, byte code, and compilers. Furthermore, you will get the most from this book if you already know ActionScript, Java, C, C#, or another language that uses similar syntax. Although we did include a chapter dedicated to covering the basics of ActionScript (the programming language that Flex applications utilize), we don't discuss any of the core APIs in detail. If you are interested in learning more about the ActionScript language, we encourage you to find a good ActionScript 3.0 book such as Essential ActionScript 3 and ActionScript 3 Cookbook.

How This Book Is Organized

We spent a lot of time organizing and reorganizing the content of this book. Although there is likely no one way to present the content that will seem perfect to all readers, we've done our best to present it in an order that we feel makes sense:

Chapter 1, Introducing Flex
: What is Flex? What are rich Internet applications (RIAs)? This chapter answers these questions, providing a context for the rest of the book.

Chapter 2, Building Applications with the Flex Framework
: In this chapter, we discuss the various elements and steps involved in building a Flex application. Topics include using the compilers, building scripts, and more.

Chapter 3, MXML
: MXML is the declarative language used by Flex. In this chapter, you'll learn the basics of MXML.

Chapter 4, ActionScript
: ActionScript is the object-oriented programming language used by Flex. In this chapter, you'll learn the basics of ActionScript 3.0.

Chapter 5, Framework Fundamentals
: Flex vastly simplifies many aspects of building applications. Although you don't often have to look under the hood, understanding the fundamentals of how the framework works is useful. In this chapter, you'll learn about Flex application life cycles, bootstrapping, and more.

Chapter 6, Managing Layout
: Flex provides many layout containers that allow you to quickly and easily create all sorts of layouts within your applications. This chapter explains how to work with those containers.

Chapter 7, Working with UI Components
: In this chapter, you'll learn about the user interface components (buttons, lists, menus, etc.) that are part of the Flex framework.

Chapter 8, Customizing Application Appearance
: Customizing the appearance of Flex applications is important because it allows you to create applications that adhere to a corporate style guide or to a creative vision. This chapter explains how to change the appearance of Flex applications.

Chapter 9, Application Components
: To make Flex application development manageable it's important to know how to break up the application into discrete parts. This chapter discusses strategies for this.

Chapter 10, Framework Utilities and Advanced Component Concepts
: Once you've learned the basics of working with components, you'll likely want to know how to expand on that knowledge. In this chapter, you'll learn about such topics as tool tips, customizing lists, pop-up windows, and more.

Chapter 11, Working with Media
: Flex allows you to include all sorts of assets and media in your applications, from images to animations to video and audio. In this chapter, you'll learn how to work with these elements.

Chapter 12, Managing State
: Flex applications and components within those applications can change from one view to another. Flex refers to these changes as *states*. Sometimes managing state is as simple as adding a new component to a form, and other times it involves changing the entire contents of the screen. How to manage state is the subject of this chapter.

Chapter 13, Using Effects and Transitions
: For animated changes between states or in response to user events or system events, Flex includes features called *transitions* and *effects*. You will learn about transitions and effects in this chapter.

Chapter 14, Working with Data
: In this chapter, you'll learn how to model data in Flex applications as well as how to link components so that they automatically update when data values change.

Chapter 15, Validating and Formatting Data
: In this chapter, you'll learn how to validate user input and how to format data such as numbers, phone numbers, and so on.

Chapter 16, Client Data Communication
: Client data communication is any transfer of data into or out of Flash Player where the data remains on the client computer. Examples of this are communication between two or more Flex applications running on the same computer, and storing persistent data on the computer. These topics are discussed in this chapter.

Chapter 17, Remote Data Communication
: In this chapter, you'll learn how to communicate from a Flex application running on a client computer to a remote data service. In the process, you'll learn how to use XML, SOAP, AMF, and more.

Chapter 18, Application Debugging
: Debugging applications is just as important as writing them. It's unusual to build an application that has no errors, and therefore it's crucial that you be able to track down those errors efficiently. In this chapter, you'll learn how to work with the debugging features of Flex.

Chapter 19, Building Custom Components
: Custom components are an important part of Flex applications because they allow you to create elements that can be used, customized, and distributed. This chapter discusses the steps necessary to create custom components using the Flex framework.

Chapter 20, Embedding Flex Applications in a Web Browser
: Many (if not most) Flex applications are deployed on the Web. That requires embedding Flex applications in web browsers. In this chapter, we talk about strategies for achieving this, as well as how to integrate Flex applications with browsers for back and forward button functionality and deep linking features.

Chapter 21, Building AIR Applications
: In this chapter, you'll learn how to use Flex to build desktop applications that run on the Adobe AIR runtime environment. This allows you to use your Flex skills to build applications that also have access to desktop-only features such as the local filesystem and system-level drag-and-drop.

Chapter 22, Building a Flex Application
: This chapter looks at the challenge of building a complete and working Flex application. In this chapter, you'll get a chance to examine different architectural challenges and possible solutions.

What You Need to Use This Book

To use this book, you should have the Flex SDK and a text editor. Our intention with this book is that those with the (free) SDK can follow along. However, we recommend that anyone who is serious about developing Flex applications use Flex Builder. If you're just starting with Flex, you might want to use the free trial version of Flex Builder initially for an optimal experience building Flex applications.

Conventions Used in This Book

The following typographical conventions are used in this book:

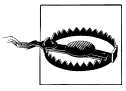
Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities

`Constant width`
: Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, and the output from commands

`Constant width bold`
: Shows commands and other text that should be typed literally by the user

`Constant width italic`
: Shows text that should be replaced with user-supplied values

This icon signifies a tip, suggestion, or general note.



This icon signifies a caution or warning.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your 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 your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming Flex 3* by Chafic Kazoun and Joey Lott. Copyright 2008 O'Reilly Media, Inc., 978-0-596-51621-5."

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

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

Comments and Questions

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:

http://www.oreilly.com/catalog/9780596516215

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:

http://www.oreilly.com

Acknowledgments

This book, perhaps more than most, represents the efforts and contributions of many people. We'd like to acknowledge the following individuals.

Many thanks are due to the many folks at O'Reilly who made this book possible. Special thanks to Steve Weiss and Audrey Doyle, not only for their hard work and patience on this book, but also for their longevity for having also seen us through the previous book. We'd also like to thank Dennis Fitzgerald for keeping us on task and as close to our deadlines as possible. We don't envy Dennis's job, since he had to push and prod us every week, but we are thankful for it. Each of these individuals has continuously gone above and beyond the call of duty, and we very much appreciate their efforts.

We'd also like to thank the many people at Adobe for working to create such a fantastic product as Flex 3, as well as for answering our questions and helping us to see what things we might have missed. We'd especially like to thank a few select people who helped with this book, or who provided content we included from the previous book: Matt Chotin, Alex Harui, Andrew Spaulding, and Manish Jethani, who not only answered our questions, but also took the time to review our chapters and provide valuable comments. We're also very grateful to Matt Horn from Adobe for graciously accepting our invitation to write the Foreword to this book.

The technical quality of this book is not due just to the work of the authors. The technical editors for this book dedicated hours and hours of time to tell us when we were wrong so that we could correct it before you read it. Therefore, we'd like to thank this book's technical editors, Romin Irani and Derek Wischusen.

From Chafic

I would first like to thank Joey. Working with him has been more than a pleasure. His experience in both the technical realm and the publishing industry, along with his patience throughout the process, were an asset to completing this book to the highest standards possible. I would also like to thank my friends, my family, and my team at Atellis for their support.

From Joey

I'd like to thank Chafic for asking me to participate in writing this book. It is an honor to work with Chafic. He is a perfectionist in the best possible way, and he sets high standards that I think show in this book. I would also like to thank my colleagues at The Morphic Group for their helpful comments on the book. And I would like to thank my friends and family for their generosity of spirit.

CHAPTER 1

Introducing Flex

Flex is a collection of technologies that enables you to rapidly build applications deployed to Flash Player, a runtime environment for delivering sophisticated user interfaces and interactivity. Flex leverages existing, matured technologies and standards such as XML, web services, HTTP, Flash Player, and ActionScript. Even though Flex allows you to create complete rich Internet and desktop applications, it does so in a relatively simple and intuitive manner. Although Flex does allow you to get under the hood for more granular control over all the elements, it significantly lowers the learning curve in that it allows you to compose applications rapidly by assembling off-the-shelf components, including UI controls, layout containers, data models, and data communication components.

In this chapter, we'll introduce Flex and Flex technologies in more detail so that you can better understand what Flex is and how you can best get started working with it. You'll learn what elements a Flex application uses and how they work together. We'll also compare and contrast Flex with other technologies for creating both standard and rich Internet applications (RIAs). Additionally, we'll review the changes and additions to Flex 3 from earlier versions.

Understanding Flex Application Technologies

If you're new to Flex, you may not yet have a clear understanding of what a Flex application is, how it works, and what benefits it has over alternative technologies and platforms. You build Flex applications utilizing the Flex framework, and you run or view them using Flash Player. In the following sections, you'll learn more about Flash Player, the Flex framework, and additional technologies that may be part of a Flex application.

Adobe has a new open source initiative, and Flex is included. As of this writing, the Flex framework and compiler are open source. You can learn more about this at *http://opensource.adobe.com/flex*.

Flash Player

Flex is part of the Adobe Flash Platform, which is a set of technologies with Flash Player at the core. Flex applications are intended to be deployed to Flash Player, meaning Flash Player runs all Flex applications. With nearly every computer connected to the Internet having some version of Flash Player installed, and an increasing number of mobile devices being Flash-enabled, Flash Player is one of the most ubiquitous pieces of software anywhere. Adobe estimates that each new version of Flash Player has adoption rates reaching 80% in less than 12 months (Flash Player 8 reached 86% within nine months). The reasons for such quick adoption rates are debatable, but there are a few factors that are almost certainly causative:

- Flash Player content is potentially more compelling and engaging than static HTML content.
- Flash Player is capable of providing integrated solutions that utilize data services, interactive UI design, media elements such as audio and video, and even real-time communications.
- Well-made Flash Player content can provide a refreshing user experience that utilizes metaphors from desktop computing, such as drag-and-drop and double-click. Flash Player frees the UI design from scrolling pages of text and images.
- Flash Player is a relatively small (one-time) download. Even with the multitude of new features added with every release, the Flash Player download is less than 1 MB. And with built-in features such as Express Install, upgrading Flash Player versions is very simple.
- Stability and security are important considerations. Flash Player is a stable program that has been around for more than a decade. Adobe is very careful with Flash Player security as well. Flash Player has very little access to the client's local system. It cannot save arbitrary files to the local system, and it cannot access Internet resources unless they meet very strict requirements.
- Flash Player is cross-platform (and cross-browser) compatible. Flash Player runs on Windows, OS X, and Linux, and on all major browsers, including Firefox, Internet Explorer, Safari, and Opera.

Flex 3 content relies on features of Flash Player 9, meaning that users must be running Flash Player 9 or later to correctly view Flex 3 content. You can read more about deploying Flex applications and detecting player versions in Chapter 20.

Using the Flex framework you can build and compile to the *.swf* format. The compiled *.swf* file is an intermediate bytecode format that Flash Player can read. Flash Player 9 has two virtual machines for running Flash and Flex content. These virtual machines are called AVM1 and AVM2. AVM1 is used to run legacy content and Flash content designed for older versions of Flash Player. AVM2 is written from the ground up, and it functions in a fundamentally different way than AVM1. With AVM2, *.swf* content is no longer interpreted. Rather, it is compiled (the equivalent of

just-in-time compilation) and run such that it can take advantage of lower-level computing power. This is very similar to how Java and .NET applications work. Flex applications always run in AVM2, meaning that Flex applications make use of the most advanced features of Flash Player.

AVM2 brings the best of both worlds. Since *.swf* content is compiled to bytecode that the ActionScript virtual machine can understand, the *.swf* format is platform-independent. That also means Flash Player ultimately dictates the functionality allowed by a Flex application. As mentioned previously, that means Flash Player can guarantee certain security safeguards so that you can deploy applications that users can trust. Yet at the same time, AVM2 compiles the content so that it runs significantly faster and more efficiently than previous versions of Flash Player.

The Flex Framework

The Flex framework is synonymous with the Flex class library and is a collection of ActionScript classes used by Flex applications. (ActionScript is the programming language used by Flash Player. We discuss it in more detail later in this chapter.) The Flex framework is written entirely in ActionScript classes and defines controls, containers, and managers designed to simplify building RIAs.

The Flex class library is the subject of much of this book. It consists of the following categories:

Form controls
: Form controls are standard controls such as buttons, text inputs, text areas, lists, radio buttons, checkboxes, and combo boxes. In addition to the standard form controls familiar to most HTML developers, the Flex class library also includes controls such as a rich text editor, a color selector, a date selector, and more.

Menu controls
: Flex provides a set of menu controls such as pop-up menus and menu bars.

Media components
: One of the hallmarks of Flex applications is rich media support. The Flex class library provides a set of components for working with media such as images, audio, and video.

Layout containers
: Flex applications enable highly configurable screen layout. You can use the layout containers to place contents within a screen and determine how they will change over time or when the user changes the dimensions of Flash Player. With a diverse set of container components you can create sophisticated layouts using grids, forms, boxes, canvases, and more. You can place elements with absolute or relative coordinates so that they can adjust correctly to different dimensions within Flash Player.

Data components and data binding
: Flex applications are generally distributed applications that make remote procedure calls to data services residing on servers. The data components consist of connectors that simplify the procedure calls, data models to hold the data that is returned, and data binding functionality to automatically associate form control data with data models.

Formatters and validators
: Data that is returned from remote procedure calls often needs to be formatted before getting displayed to the user. The Flex class library includes a robust set of formatting features (format a date in a variety of string representations, format a number with specific precision, format a number as a phone number string, etc.) to accomplish that task. Likewise, when sending data to a data service from user input, you'll frequently need to validate the data beforehand to ensure that it is in the correct form. The Flex class library includes a set of validators for just that purpose.

Cursor management
: Unlike traditional web applications, Flex applications are stateful, and they don't have to do a complete screen refresh each time data is sent or requested from a data service. However, since remote procedure calls often incur network and system latency, it's important to notify the user when the client is waiting on a response from the data service. Cursor management enables Flex applications to change the cursor appearance to notify the user of such changes.

State management
: A Flex application will frequently require many state changes. For example, standard operations such as registering for a new account or making a purchase usually require several screens. The Flex class library provides classes for managing those changes in state. State management works not only at the macro level for screen changes, but also at the micro level for state changes within individual components. For example, a product display component could have several states: a base state displaying just an image and a name, and a details state that adds a description, price, and shipping availability. Furthermore, Flex provides the ability to easily apply transitions so that state changes are animated.

Effects
: Flex applications aren't limited by the constraints of traditional web applications. Since Flex applications run within Flash Player, they can utilize the animation features of Flash. As such, the Flex class library enables an assortment of effects, such as fades, zooms, blurs, and glows.

Deep linking and browser back and forward button integration
: The browser integration features of Flex allow for deep linking (unique URLs for different application states allowing for linking directly to a state) as well as allowing the browser's back and forward buttons to correctly navigate through states of the Flex application.

Drag-and-drop management
: The Flex class library simplifies adding drag-and-drop functionality to components with built-in drag-and-drop functionality on select components and a manager class that allows you to quickly add drag-and-drop behaviors to components.

Tool tips
: Use this feature of the Flex class library to add tool tips to elements as the user moves the mouse over them.

Style management
: The Flex class library enables a great deal of control over how nearly every aspect of a Flex application is styled. You can apply style changes such as color and font settings to most controls and containers directly to the objects or via Cascading Style Sheets (CSS).

Localization
: Using Flex you can localize applications by way of the resource management part of the Flex framework. This allows you to use resource bundles containing localized text, images, and other resources.

Flex Builder 3

Flex Builder 3 is the official Adobe IDE for building and debugging Flex applications. Built on the popular Eclipse IDE, Flex Builder has built-in tools for writing, debugging, and building applications using Flex technologies such as MXML and ActionScript.

The Flex framework ships as part of Flex Builder. Flex Builder and the Flex framework, however, are not synonymous. You don't have to use Flex Builder to use the Flex framework. Instead, you can opt to install the free Flex SDK, which includes the compiler and the Flex framework. You can then integrate the Flex framework with a different IDE, or you can use any text editor to edit the MXML and ActionScript files, and you can run the compiler from the command line.

Flex Builder is a commercial product. See *http://www.adobe.com/go/flexbuilder* for more information.

Integrating with Data Services

Data services are an important aspect of most Flex applications. They are the way in which the Flex application can load and send data originating from a data tier such as a database [we discuss the concept of tiers in "The Differences Between Traditional and Flex Web Applications" later in this chapter]. Flash Player supports any text data, XML, a binary messaging format called AMF, and persistent socket connections, allowing for real-time data pushed from the server to the client.

Each data format that Flex supports may or may not require special server resources. For example, a Flex application can request XML data from a static resource or from a dynamic resource such as a PHP page. AMF is a binary messaging format that Flash Player understands natively, but for a server to interact with Flash Player via AMF it requires an AMF translator on the server, such as the remote object services that are part of LiveCycle Data Services.

Flex simplifies working with data services by way of classes and components that are part of the framework. We discuss working with data services in more detail in Chapter 17.

Integrating with Media Servers

Since Flex applications are deployed using Flash Player, they can leverage the media support for Flash video and audio. Although Flash Player can play back Flash video and MP3 audio as progressive downloads, you can benefit from true streaming media by way of a technology such as Flash Media Server (*http://www.adobe.com/go/fms*) or Red5 (*http://www.osflash.org/red5*).

Additional Flex Libraries and Components

Additional Flex libraries and components are available for you to use when building Flex applications. The most obvious of these are the additional components available with the Professional edition of Flex Builder 3. The advanced data grid and the charting component set are available only with the Professional edition of Flex Builder 3, and they're not included with the free Flex SDK or with the Standard edition of Flex Builder 3.

However, the advanced data grid and charting components are far from the only examples of add-on libraries and components for Flex. There are many such examples, available both commercially and free. A good list of available libraries and components is available at *http://www.flex.org/components*.

Add-on libraries enable more rapid application development because they provide prebuilt functionality. For example, with the addition of the charting component set you can quickly and simply add robust charting and graphing features to Flex applications.

Using Flex Elements

The Flex framework includes a core set of languages and libraries that are the basis of any Flex application. Using MXML, ActionScript, and the Flex class library you can construct and compile *.swf* content that you can then deploy to Flash Player.

MXML

MXML is an XML-based markup language primarily for describing screen layout. In that respect, it is much like HTML. Using MXML tags you can add components such as form controls and media playback components to layout containers such as grids.

In addition to screen layout, you can use MXML to describe effects, transitions, data models, and data binding. MXML is robust enough that it is possible to build many applications entirely with MXML. Flex Builder enables you to construct MXML with a WYSIWYG approach, enabling you to build basic Flex applications without writing any code.

Although the WYSIWYG approach is helpful for basic prototypes and simple applications, writing MXML code is still necessary for more complex tasks. Additionally, sophisticated Flex applications generally require both MXML and ActionScript.

MXML is a declarative way to create Flex content, but the simplicity should not fool you into thinking that MXML is not powerful. MXML provides a fast and powerful way to create layout and UI content. However, MXML documents get compiled in several steps, the first of which converts the MXML to an ActionScript class. This means MXML documents provide you with all the power of object-oriented design, but with the convenience of a markup language. Furthermore, MXML documents are treated as ActionScript classes at runtime.

ActionScript

ActionScript is the programming language understood by Flash Player and is the fundamental engine of all Flex applications. MXML simplifies screen layout and many basic tasks, but all of what MXML does is made possible by ActionScript, and ActionScript can do many things that MXML cannot do. For example, ActionScript is necessary to respond to events such as mouse clicks.

Although it is possible to build an application entirely with MXML or entirely with ActionScript, it is more common and more sensible to build applications with the appropriate balance of both MXML and ActionScript. Each offers benefits, and they work well together. MXML is best suited for screen layout and basic data features. ActionScript is best suited for user interaction, complex data functionality, and any custom functionality not included in the Flex class library.

ActionScript is supported natively by Flash Player and does not require any additional libraries to run. All the native ActionScript classes (classes that are built into Flash Player) are packaged in the `flash` package or in the top-level package. In contrast, the Flex framework is written in ActionScript, but those classes are included in an *.swf* file at compile time. All the Flex framework classes are in the `mx` package.

You can learn much more about ActionScript in Chapter 4. In fact, if any of the terms we've used in this section are unfamiliar to you, you will have an opportunity to learn more about them in that chapter.

Working with Data Services (Loading Data at Runtime)

Flex applications are generally distributed applications. That means several computers work in conjunction to create one system. For example, all Flex applications have a client tier (discussed shortly) that runs on the user's computer in the form of an *.swf* running in Flash Player. In most cases, the client tier communicates with a server or servers to send and retrieve data. The servers provide what are called *data services*, which are essentially programs that have public interfaces (APIs) whereby a client can make a request to a method of that program. When a client makes such a request, it's called a *remote procedure call*, or RPC.

There are many types of data services. In its simplest form a data service could consist of a static text file or XML document served from a web server. A slightly more sophisticated data service might be a dynamic XML document generated via a server-side script or program, such as a PHP or ASPX page. Many data services require greater sophistication. One of the most common types of such a sophisticated data service is the web service. Web services use XML (generally in the form of SOAP) as a messaging format, and they enable RPCs using HTTP for requests and responses. Although a SOAP web service is an example of a standards-based data service, many types of data services don't necessarily conform to a particular standard set by the W3C. Many programs on the Web, for example, expose primitive data services that use arbitrary messaging formats and protocols. One such program is used by MapQuest, a popular mapping web site. For instance, you would use the following URL to view a MapQuest page with a map of Los Angeles:

http://www.mapquest.com/maps/map.adp?country=US&city=Los+Angeles&state=CA

Notice that the query string uses arbitrary parameters to determine what to map. Therefore, if you wanted to display a map of New York, you would change the city and state parameter values in the URL as follows:

http://www.mapquest.com/maps/map.adp?country=US&city=New+York&state=NY

Flash Player is capable of making RPCs to many types of data services. For example, Flash Player can make requests to any web resource using HTTP, which means it can make requests to many primitive data services such as a static or a dynamic XML document, or the MapQuest example mentioned previously. That also means it can make requests to web services. Moreover, the Flex class library simplifies requests to most data services.

In addition to the types of data services previously mentioned, Flex applications can also make calls to methods of classes on the server, using a technology called

Remoting. Remoting uses a binary messaging format called *AMF*, which is supported natively by Flash Player. AMF has all the benefits of SOAP, but since it is binary, the bandwidth overhead is greatly reduced. And since AMF is natively supported by Flash Player, no special coding is necessary to use Remoting data services from the client tier. However, for a Remoting data service to be available to the client tier, it must be made accessible via a piece of software that resides on the server and can read and write AMF packets as well as delegate the requests to the correct services. You can find a list of Remoting server products in Chapter 17.

The Differences Between Traditional and Flex Web Applications

Many applications deployed on the Web use HTML as the user interface. Flex applications are similar in many respects, but they have distinct differences. If you're used to building applications that use an HTML UI, it's important to take a few moments to shift how you approach building applications when you start working with Flex. What works for HTML-based applications may or may not work for Flex applications.

Both traditional and Flex applications are generally *n*-tiered. The exact number and types of tiers an application has depend on many factors. Most traditional applications have, at a minimum, a data tier, a business tier, and a presentation tier. Flex applications have a data tier and a business tier; however, as noted earlier, they also introduce a client tier, which is what strongly differentiates them from traditional web applications. The client tier of Flex applications enables clients to offload computation from the server, freeing up network latency and making for responsive and highly interactive user interfaces.

Data tiers generally consist of databases or similar resources. Business tiers consist of the core application business logic. As an example, a business tier may accept requests from a client or presentation tier, query the data tier, and return the requested data.

In traditional applications, the presentation tier consists of HTML, CSS, JavaScript, JSP, ASP, PHP, or similar documents. Typically a request is made from the user's web browser for a specific presentation tier resource, and the web server runs any necessary interpreters to convert the resource to HTML and JavaScript, which is then returned to the web browser running on the client computer. Technically the HTML rendered in the browser is a client tier in a traditional web application. However, since the client tier of a traditional web application is stateless and fairly unresponsive, it is generally not considered a full-fledged tier. (The exception to that generalization is the case of Ajax applications, which use client-side JavaScript and XML to build responsive and sophisticated client tiers.)

Flex applications generally reside embedded within the presentation tier. In addition, Flex applications can integrate with the presentation tier to create tightly coupled client-side systems. Flex applications use Flash Player to run sophisticated client-tier portions

of the application. The Flex application client is *stateful*, which means it can make changes to the view without having to make a request to the server. Furthermore, the Flex application client is responsive. For example, Flash Player can respond to user interaction such as mouse movement, mouse clicks, and keyboard presses, and it can respond to events such as notifications from the business tier when data is returned or pushed to the client. Flash Player also can respond to timer events. Since Flash Player is a smart client, it is capable of saving on network overhead and bandwidth usage by managing client-side logic without having to consult the business tier. For example, Flex applications can walk the user through a step-based or wizard-like interface, collect and validate data, and allow the user to update and edit previous steps, all without having to make requests to the business tier until the user wants to submit the data. All of this makes Flex clients potentially far more compelling, responsive, and engaging than traditional web applications.

Because the Flex application client tier is so much more sophisticated than the presentation tier of a traditional web application, the Flex client tier requires significantly more time and resources to build successfully. A common mistake is to assume that Flex client tiers require the same time and resources as a traditional web application presentation tier. Successful Flex client tiers often require the same time and resources during design, implementation, and testing phases as the business tier.

Understanding How Flex Applications Work

Flex applications deployed on the Web work differently than HTML-based applications. It's important to understand how Flex applications work in order to build them most effectively. When you understand how Flex applications work, you can know what elements are necessary for an application and how to build the application for the best user experience. Figure 1-1 summarizes the basic concepts discussed in this section.

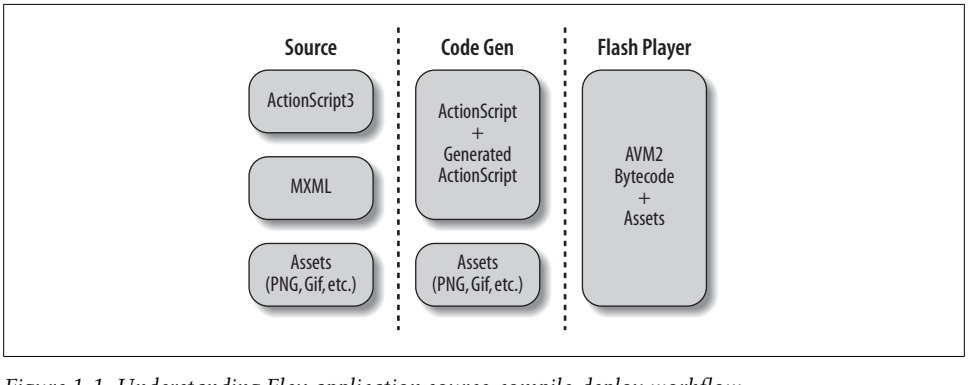


Figure 1-1. Understanding Flex application source-compile-deploy workflow

Every Flex application deployed on the Web utilizes Flash Player as the deployment platform. That means a fundamental understanding of Flash Player is essential to understanding Flex. Additionally, all Flex applications use the Flex framework at a minimum to compile the application. As such, it's important to understand the relationship between the source code files, the compiler, and Flash Player.

All Flex applications require at least one MXML file or ActionScript class file, and most Flex applications utilize both MXML and ActionScript files. The MXML and ActionScript class files comprise the source code files for the application. Flash Player does not know how to interpret MXML or uncompiled ActionScript class files. Instead, it is necessary to compile the source code files to the *.swf* format, which Flash Player can interpret. A typical Flex application compiles to just one *.swf* file. You then deploy that one *.swf* to the server, and when requested, it plays back in Flash Player. That means that unlike HTML-based applications, the source code files remain on the development machine, and you do not deploy them to the production server.

Asset files such as MP3s, CSS documents, and PNGs can be embedded within an *.swf* or they can be loaded at runtime. When an asset is embedded within an *.swf*, it's not necessary to deploy the file to the production server, since it is compiled within the *.swf* file. However, since embedding assets within the *.swf* often makes for a less streamlined downloading experience and a less dynamic application, it is far more common to load such assets at runtime. That means the asset files are not compiled into the *.swf*, and much like an HTML page the assets are loaded into Flash Player when requested by the *.swf* at runtime. In that case, the asset files must be deployed to a valid URL when the *.swf* is deployed.

Data services are requested at runtime. That means the services must be available at a valid URL when requested at runtime. For example, if a Flex application utilizes a web service, that web service must be accessible from the client when requested. Media servers and Flex Enterprise Services must also be accessible when utilized by Flex applications.

Understanding Flex and Flash Authoring

Many developers first learning about Flex 3 may still be unclear as to the relationship between Flex and Flash authoring, the traditional tool for creating content for Flash Player. First, you do not have to understand Flash authoring in order to work with Flex. In fact, you can get started with Flex without any prior knowledge of or experience in Flash authoring.

Flash authoring is a product that was developed in 1996 as a vector animation tool primarily aimed at creating animation content for the Web. In the many versions since that time, both Flash authoring and Flash Player (the deployment platform for Flash authoring content) have enabled greater and greater capabilities, and developers began to create RIAs with the tools. However, although Flash authoring is a fantastic tool for

creating animations, it is not the ideal tool for creating applications. The metaphors that Flash authoring uses at its core (such as timelines) are simply not applicable to application development.

Flex 3 is a product aimed primarily at creating applications. The framework includes a rich set of layout and user interface components, and the technology uses metaphors such as states and transitions that are appropriate to application development.

Both Flex and Flash authoring allow you to create *.swf* content that runs in Flash Player. In theory, you can achieve the same things using both products. However, Flash is a tool that allows you to create timeline-based animations and to use drawing tools best suited for expressiveness, whereas Flex allows you to much more rapidly assemble screens of content with transitions and data communication behaviors. As with any craft, it is advisable to use the best tool for the job. Until now, Flash authoring was one of the only tools for creating *.swf* content. But with Flex 3, we now have a tool with a more specific focus.

Although many people may initially try to frame the Flex and Flash authoring debate as a winner-takes-all scenario, it's rather naïve to think of them as competing technologies. Rather, they are two complementary technologies that allow all Flash platform developers to utilize specialized tools when creating rich Internet content. In fact, Flex and Flash authoring can work very well together. As you'll see in this book, Flex can import content created in Flash authoring, allowing you to create RIAs that utilize timeline-based content.

What's New in Flex 3

If you are familiar with earlier versions of Flex (Flex 1, Flex 1.5, and Flex 2), you may be interested in the relationship between Flex 3 and those earlier versions and what is new in Flex 3. In this section, we'll look at this subject in more detail.

If you're familiar with Flex 1 or Flex 1.5, but you haven't used Flex 2, you will find that Flex 3 is dramatically different from the version or versions of Flex you have used. Although Flex 3 continues to utilize MXML and ActionScript (both supported in Flex 1 and 1.5), it is vastly different from Flex 1 and Flex 1.5 in other respects. Flex 3 allows you to compile and deploy independent *.swf* files without any sort of expensive server-side services as was required by Flex 1 and 1.5. Flex 3 requires Flash Player 9, which allows for (and requires) the use of ActionScript 3. This latest ActionScript version introduces significant changes to the Flash Player API that enable a much improved way to add and remove display objects (including components) to the view.

If you've used Flex 2, but you're new to Flex 3, you may be interested in knowing what is new in Flex 3 that wasn't available in Flex 2. Some of the most significant changes/additions to Flex 3 are as follows:

Runtime localization/internationalization
: Although Flex 2 had built-in localization features, they were compile-time only, and they didn't allow for runtime switching of locales. In Flex 3, the locale can be changed at runtime, and the resource bundles can even be downloaded on demand.

Flex framework caching
: Flex 3 allows you to use the new Flash Player caching feature to cache the Flex framework, reducing the file size for Flex *.swf* files.

Support for Adobe AIR
: Adobe AIR is a runtime environment that allows Flex developers to build applications for the desktop. Flex 3 has native support for building AIR applications.

You can learn about these new features and more throughout the book.

Summary

In this chapter, we introduced the basics of what Flex is and what technologies and products are used to create Flex applications. You learned that Flex 3 consists of a framework (a class library) and a compiler that allow you to rapidly create Flex applications. These applications are *.swf* files that you can then run in Flash Player 9.

CHAPTER 2

Building Applications with the Flex Framework

The majority of this book is dedicated to programming Flex applications, with detailed discussions of working with MXML and ActionScript. However, to meaningfully use most of what we discuss in the chapters that follow, you'll need to know how to create a Flex project, how to compile the project, and how to deploy that project so that you can view it.

In this chapter, we'll discuss important topics like the tools needed to create Flex applications and how to create new projects for Flex applications. We'll look at elements comprising a Flex project and discuss compiling and deploying Flex applications.

Using Flex Tool Sets

To work with Flex and build Flex applications, you'll need tools. At a minimum, you must have a compiler capable of converting all your source files and assets into the formats necessary to deploy the application. That means you need to be able to compile MXML and ActionScript files into an *.swf* file.

There are two primary tools you can use that include the necessary compilers:

- The Flex Software Development Kit (SDK)
- Flex Builder 3

The Flex SDK is a free product that includes the entire Fl`ex framework as well as the `mxmlc` and `compc` compilers (see "Building Applications later in this chapter for more details on the compilers). Download the SDK at *http://www.adobe.com/products/flex/flexdownloads/*.

Flex Builder 3 is a professional IDE designed for Flex development, and it too includes the `mxmlc` and `compc` compilers. You can download a trial version of Flex Builder 3 or purchase a license at *http://www.adobe.com/go/flex*.

Flex Builder includes the entire copy of the SDK. Beginning with Flex Builder 3, it contains support for targeting different versions of the SDK. You can find the different versions of the SDK in *<Flex Builder Install Folder>\sdks*.

You can work with Flex Builder 3 in two ways: as a standalone application and as a plug-in for Eclipse. The standalone version of Flex Builder 3 is built on Eclipse, so it and the plug-in version are essentially equivalent. The primary differences are:

- Flex Builder 3 standalone does not require that you already have Eclipse installed, making it an optimal solution for those who have no other use for Eclipse. On the other hand, if you already use Eclipse, or if you intend to use Eclipse for other purposes, the standalone version would essentially require you to have two installations of Eclipse—one running Flex Builder and one standard installation. If you use or plan to use Eclipse for other reasons, you should definitely install the plug-in version of Flex Builder 3.
- The standalone version disables Java Development Tools (JDT), a plug-in used by some standard Eclipse features such as Ant. If you want to use JDT, you should install the plug-in version of Flex Builder 3.

Since Flex Builder is built on Eclipse, you can use any third-party Eclipse plug-ins with the standalone version of Flex Builder.

Many factors might drive your decision as to whether to use the Flex SDK or Flex Builder 3. The following is a list of just a few to consider:

Price
: The Flex SDK is a free product. It includes the entire Flex framework. Flex Builder 3, on the other hand, is a commercial product. There is no difference in price between the standalone and plug-in versions of Flex Builder 3.

Commitment to an existing IDE
: If you already have a considerable investment in an IDE in terms of time and resources, and if that IDE works very well for you, you may want to integrate the Flex SDK with your existing IDE. On the other hand, if you're already using Eclipse, consider that you can install the Flex Builder 3 plug-in for an existing installation of Eclipse.

Debugging capabilities
: The Flex SDK includes a command-line debugger. However, Flex Builder 3 includes an integrated debugger that allows you to set breakpoints and step through code, all from within your IDE.

Efficiency
: Unless and until other IDEs have increased support for Flex (ActionScript and MXML), Flex Builder is the fastest way to build Flex applications. With its built-in code hinting, code completion, error detection, and debugging capabilities, Flex Builder is far superior to the SDK for serious Flex application developers.

The majority of the content of this book is not dependent on any one tool. Much of our focus is on working with the Flex framework and ActionScript 3.0 and will require only the Flex SDK. When there are specific topics that do have dependencies on a particular tool, we make that clear. For example, in this chapter we discuss the differences between configuring a Flex Builder project versus a Flex SDK project.

Creating Projects

A Flex application consists of potentially many files. Although it's possible that a Flex project could consist of as little as one source file, most use tens if not hundreds of files. A typical Flex project might utilize the following:

MXML files
: These files contain the majority of the application view—the layout and UI components. You can read an introduction to MXML in Chapter 3. You can also learn about application and MXML components (both written in MXML) in Chapter 9.

ActionScript classes
: These files contain the source code for all the custom components, data models, client-side business logic, and server proxies. You will find an introduction to ActionScript in Chapter 4.

XML files
: Although XML is frequently loaded from a server as a dynamic response to an HTTP request from Flash Player, many applications also utilize static XML files as configuration parameters.

Image files
: Flex applications can embed image files or load them at runtime. We cover working with images in Chapter 11.

Audio and video files
: Flex applications can load audio and video content for playback within the application. Audio and video are almost always loaded at runtime. We discuss audio and video in Chapter 11.

Runtime shared libraries
: Runtime shared libraries are *.swf* files that contain code libraries that are shared between two or more Flex applications deployed on the same domain. To utilize a runtime shared library, you need two files: an *.swf* and an *.swc*. The *.swf* file contains the libraries, and the *.swc* file is used by the compiler to determine which

libraries to exclude from the application *.swf*. We discuss runtime shared libraries in more detail in Chapter 20.

HTML wrapper file
: Flex applications are deployed on the Web or as an AIR application. The HTML wrapper file is used when deploying to the Web. The published application is an *.swf* file. The most common way to play back an *.swf* on the Web is to embed it in an HTML page and execute it with Flash Player. The HTML wrapper file is the file that embeds the *.swf*.

Setting Up a New Project

How you configure a new Flex project depends in large part on what tool set you are using. If you're using the Flex SDK, that tool set generally requires the most work to configure a new project. We'll first discuss creating a project using the SDK. If you only ever intend to use Flex Builder to create and work with projects, you can go ahead and skip to "Creating a Flex Builder 3 project."

Creating an SDK project

Presumably, if you're using the Flex SDK, you're integrating it with an IDE such as Eclipse (*http://www.eclipse.org*), PrimalScript (*http://www.sapien.com*), or FlashDevelop (*http://www.flashdevelop.org*). If you are indeed using an IDE, you most likely want to start a new project (or workspace or whatever particular terminology your IDE uses). If you are not using an IDE (you like to edit code using a plain-text editor), you will want to create a new directory for the project.

You'll place all the project files in the project directory, likely organizing them into subdirectories. Which subdirectory structure you use is ultimately up to you. You'll need to know where and how you're organizing all the source code and assets so that you can configure the appropriate compiler options when building the application. (We discuss compiler options in "Building Applications," later in this chapter.) Files typically are organized into the following directories:

src
: A directory containing all the source MXML and ActionScript class files. The files are then generally organized into packages. You can organize both MXML and ActionScript files into packages in a Flex project. We discuss packages in more detail in Chapter 3 and Chapter 4.

bin
: A directory to which you save the compiled version of the application.

html
: A directory in which you keep the HTML wrapper file(s).

src/assets_embed
: A directory in which you save all the asset files embedded by the application at compile time. Assets that will be loaded at runtime should be placed in another directory where your application can access them at runtime.

build
: A directory in which you can place build scripts if using Apache Ant.

Creating a Flex Builder 3 project

With Flex Builder 3, you can easily create a new project. From the Flex Builder menu select File→New→Flex Project to open the Flex Project dialog and follow these steps:

1. Step 1 asks for a project name, location, application type, and server technology. The application type specifies if it's a web application that generates an *.swf* file to run in a browser or a Flex application for Adobe AIR runtime. For server technology, all examples in this book will work via the None option (Figure 2-1).

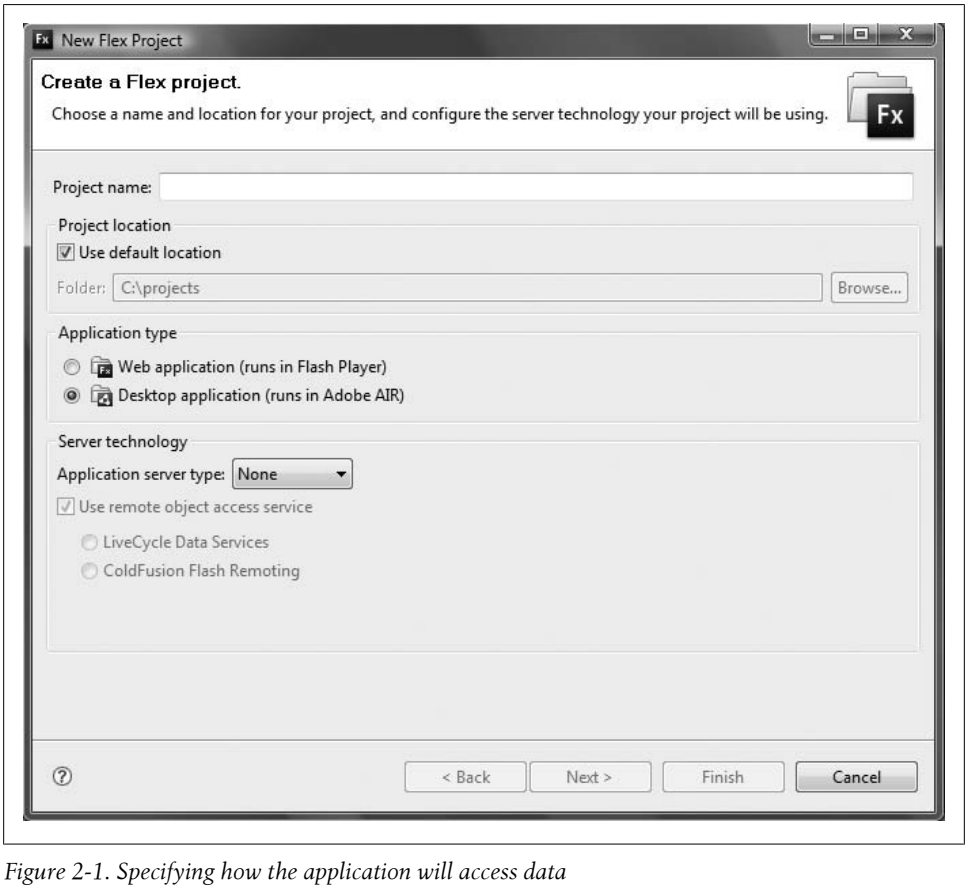


Figure 2-1. Specifying how the application will access data

2. At the completion of step 1, you can click Finish. If you click Next, you'll see a second step asking you to customize the source path and library path. These settings specify classes and libraries that you want to use but that reside outside the project directory or in a nonstandard location within the project directory. Unless stated otherwise, no examples in this book require you to customize the source path or library path. See Figure 2-2.

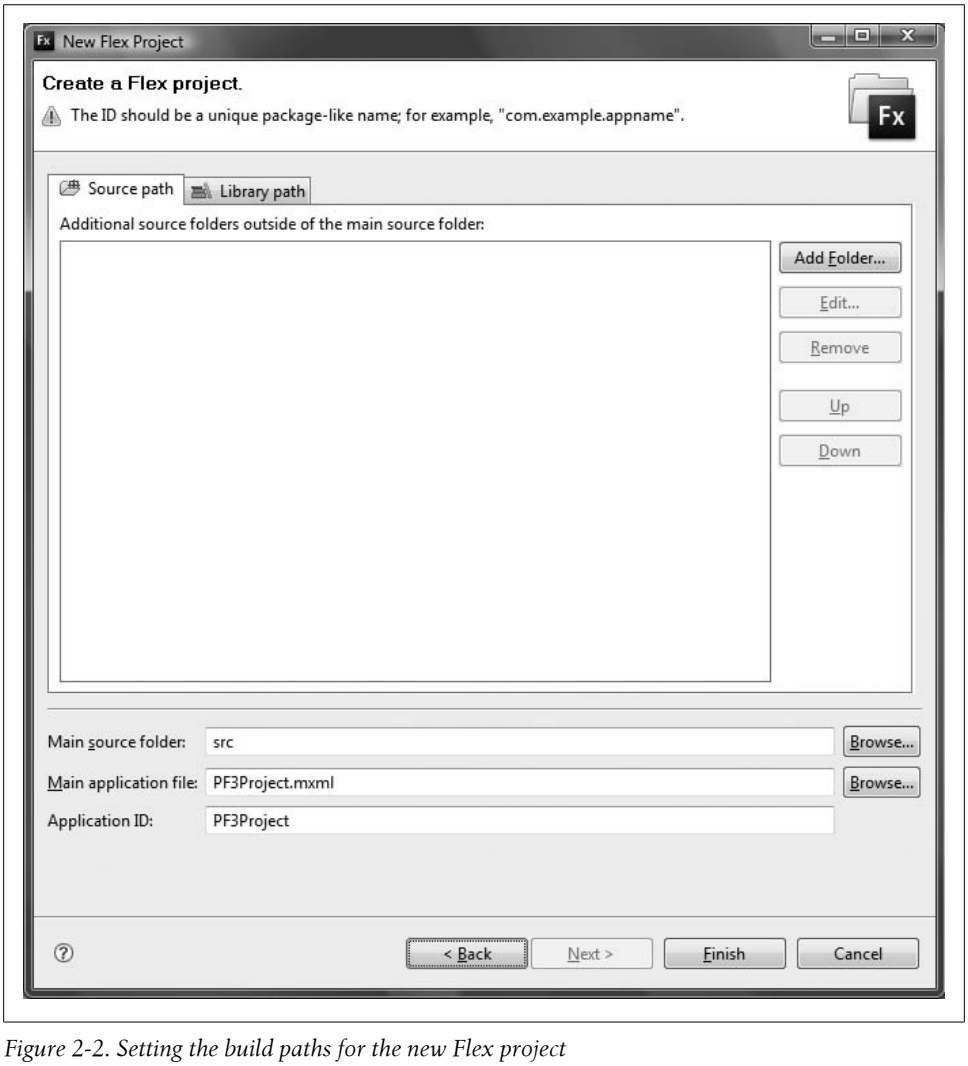


Figure 2-2. Setting the build paths for the new Flex project

When you create a Flex Builder project, you'll see that the new directory has a *bin-debug* directory to which Flex Builder saves the compiled application by default, as well as an *html-template* directory that stores the templates used by Flex Builder to generate

the HTML wrapper file. You'll also see that the new project automatically created an MXML document with the same name as the project within the *.src* directory by default.

Building Applications

Once you've created a project and written some code, you'll want to build the project, which means compiling it and deploying it. How you accomplish these tasks depends, in part, on what tools you're using. The following sections discuss how to compile using the `mxmlc` compiler. If you're using Flex Builder, you may want to skip directly to "Compiling Using Flex Builder" later in this chapter, although it is always good to know about `mxmlc`, especially if you intend to use Ant or any build tool.

Compiling Using mxmlc

The `mxmlc` compiler is used to compile Flex applications (versus `compc`, which is used to compile components and libraries). When you use Flex Builder to compile, it automatically calls `mxmlc` (Flex Builder includes the SDK).

There are several ways you can use `mxmlc`, including from the command line, from a *.bat* or shell script, from an IDE, and from Apache Ant. Initially, we'll look at using `mxmlc` from the command line since it's the most basic way to use the compiler (though we'll also look at using the compiler via Apache Ant later in this chapter). The compiler flags we'll look at from the command line also apply to any other use of the compiler.

Configuring for Windows

When you want to work with `mxmlc` from the command line, it's generally a good idea to make sure you add it to your system path. If you're running Windows and you're uncertain how to edit your system path, follow these steps:

1. Right-click My Computer from the desktop or from the Start menu, and select Properties.
2. Click the Advanced tab, and then click the Environment Variables button.
3. In the System Variables list in the bottom part of the dialog, scroll until you see a variable called Path. Then edit the variable either by double-clicking on it or by selecting it and then clicking the Edit button.
4. At the end of the existing value, add the path to the Flex SDK's *bin* directory. If you're using Flex Builder, the default location is *C:\Program Files\Adobe\Flex Builder 3\sdks\<sdk version>*. If you're using the SDK and you installed the SDK in *C:\FlexSDK*, the location is *C:\FlexSDK\bin*. Windows uses a semicolon (`;`) as a delimiter. If necessary, add a semicolon between the existing value and the new addition.
5. Click OK on each open dialog.

Configuring for OS X and Linux

For OS X and Linux, you'll want to set the `PATH` environment variable in your shell. If you are using *.bash* or any shell that supports *.profile* files, you will want to add a *.profile* file in your user directory (or edit the file if it already exists). You can edit the file with any text editor that you want. If you are familiar with *vi*, for example, you can simply open a Terminal and type **`vi ~/.profile`**. The *.profile* should contain a line such as the following:

```
export PATH=$PATH:/Users/username/FlexSDK/bin
```

The preceding line of code assumes that you have installed the SDK in your user directory (you'll need to change *`username`* to your actual username). If you've installed the SDK elsewhere, you should modify the path correspondingly. Also note that the preceding code assumes that you don't want to add additional directories to your path. If you have an existing *.profile* file that already contains an `export PATH` line, you should simply append the Flex *bin* path to that line using a colon (`:`) as a delimiter. For example:

```
export PATH=$PATH:/existing/directories:/Users/username/FlexSDK/bin
```

Once you've edited the *.profile* you'll need to run the following command from any existing Terminal window or command prompt:

```
source ~/.profile
```

Beginning with the command line

To use the compiler from the command line you simply specify the compiler name followed by the options. The only required option is called `file-specs`, and it allows you to specify the entry point to the application you want to compile, that is, the main MXML document (or ActionScript class):

```
mxmlc -file-specs SampleApplication.mxml
```

Notice that `file-specs` is preceded by a hyphen. All options are preceded by a hyphen.

You can get help for the compiler by running `mxmlc` with the help option:

```
mxmlc -help
```

The `file-specs` option is the default option for `mxmlc`. That means a value that is not preceded by an option flag will be interpreted as the value for `file-specs`. The following example is equivalent to the preceding example:

```
mxmlc SampleApplication.mxml
```

The examples that follow attempt to compile *SampleApplication.mxml* to *SampleApplication.swf*.

Specifying an output location

By default, `mxmlc` compiles the application to an *.swf* with the same name as the input file (i.e., *SampleApplication.mxml* compiles to *SampleApplication.swf*) in the same directory as the input file. However, you can specify an output path and *.swf* name using the `output` option. The following compiles *SampleApplication.mxml* to *bin/main.swf*:

```
mxmlc SampleApplication.mxml -output bin/main.swf
```

Specifying source paths

The source path is the path in which the compiler looks for required MXML and ActionScript files. By default, the compiler looks in the same directory as the compile target (the file specified by `file-specs`). This means it will also look in subdirectories for documents and classes that are in packages. However, any files located outside the same directory structure won't be found using the default source path compiler settings.

You can use the `source-path` option to specify one or more directories in which the compiler should look for the MXML and ActionScript files. You can specify a list of directories by using spaces between directories. The following example looks for files in the current directory as well as in *C:\FlexApplicationCommonLibraries*:

```
mxmlc -source-path . C:\FlexApplicationCommonLibraries -file-specs
SampleApplication.mxml
```

Customizing the application background color

The default background color is the blue you see for most Flex applications. Use the `default-background-color` option to customize the background value. You can specify the value using 0x-prefixed hexadecimal representation in the form of RRGGBB. Use this in cases where you customize the appearance of an application and want the initial color seen by the user to match the overall look of your application. The following sets the default background color of *SampleApplication* to white:

```
mxmlc -default-background-color=0xFFFFFF SampleApplication.mxml
```

Note that the background color in this case is the background color of Flash Player. A Flex application being deployed to the Web has several places where its background is set. There is the background color in the HTML document, the *.swf* file, and the Flex root container. Setting the `-default-background-colorph` compiler setting will set only the background color of the *.swf* file. The most common way to set the background color for all three values is to set the root `Application` tag's `backgroundProperty` style. Setting the value of this style will instruct the Flex compiler to set the values of the root container (`Application`), the *.swf* file, and the HTML document background if you are using the provided templates.

Changing script execution settings

Flash Player automatically places restrictions on script execution in an attempt to prevent applications from crashing client systems. This means that if too many levels of recursion occur, or if a script takes too long to execute, Flash Player will halt the script.

The `default-script-limits` option allows you to customize each of these settings. The option requires two values: one for the maximum level of recursion and one for the maximum script execution time. The default maximum level of recursion is 1000, and the default maximum script execution time is 60 seconds (you cannot specify a value larger than 60 for this parameter):

```
mxmlc -default-script-limits 200 15 -file-specs SampleApplication.mxml
```

Although it's important to know about the existence of `default-script-limits`, it's also important to know that it should rarely be used. If you have to increase the `default-script-limits` setting for an application to avoid an error, frequently it's because there is a problem in the code or in the application logic.

Setting metadata

The *.swf* format allows you to encode metadata in the application file. The allowable metadata includes the following: `title`, `description`, `creator`, `publisher`, `language`, and `date`. You can set these values using options with the same names as the metadata elements:

```
mxmlc -title "Sample Application" -description "A Flex Sample Application" -file-
specs SampleApplication.mxml
```

Using incremental builds

By default, when you compile from the command line, `mxmlc` compiles a clean build every time. That means that it recompiles every source file, even if it hasn't changed since you last compiled. That is because by default, `mxmlc` doesn't have a way of knowing what has changed and what hasn't.

There are times when a clean build is exactly the behavior you want from `mxmlc`. However, in most cases you'll find that it's faster to use *incremental builds*. An incremental build is one in which the compiler recompiles only those elements that have changed since you last compiled. For all other elements it uses the previously compiled versions. Assuming that not much has changed since the previous compile, an incremental build can be much faster than a clean build.

If you want to use incremental builds, you need a way to determine what things have changed between builds. When you set the `-incremental` option to `true`, `mxmlc` writes to a file in the same directory as the target file you are compiling, and it shares the same name. The name of the cache file is *TargetFile_<#>.cache*, in which the # is a number

generated by the compiler. For example, the following might write to a file called *SampleApplication_302345.cache* (where the number is determined by the compiler):

```
mxmlc -incremental=true -file-specs SampleApplication.mxml
```

Storing compiler settings in configuration files

Although it is undoubtedly great fun to specify compiler options on the command line, you can also store settings in configuration files. You can then specify the configuration file as a single option from the command line. The `load-config` option lets you specify the file you want to load to use as the configuration file:

```
mxmlc -load-config=configuration.xml SampleApplication.mxml
```

By default, `mxmlc` uses a configuration file called *flex-config.xml* located in the *frameworks* directory of the SDK or Flex Builder installation. If you specify a value for the `load-config` option, that can override *flex-config.xml*. Many, though not all, of the settings in *flex-config.xml* are required. That means it's important that you do one of the following:

- Copy and modify the content of *flex-config.xml* for use in your custom configuration file. When you do so, you will likely have to modify several values in the file so that they point to absolute paths rather than relative paths. Specifically, you have to modify:
  - The `<external-library-path>` setting from the relative *libs/playerglobal.swc* to a valid path pointing to the actual *.swc* file
  - The `<library-path>` settings from `libs` and `locale/{locale}` to the valid paths pointing to those resources (you can keep the `{locale}` variable)
- Load your custom file in addition to the default. When you use the `=` operator to assign a value to the `load-config` option, you load the file in place of the default. When you use the `+=` operator, you load the file in addition to the default. Any values specified in the custom configuration file override the same settings in the default file:

  ```
  mxmlc -load-config+=configuration.xml SampleApplication.mxml
  ```

Configuration files must have exactly one root node, and that root node must be a `<flex-config>` tag. The `<flex-config>` tag should define a namespace, as in the following example:

```
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
</flex-config>
```

Within the root node you can nest nodes corresponding to compiler options. You can configure any and every compiler option from a configuration file. However, the option nodes must appear in the correct hierarchy. For example, some option nodes must appear within a `<compiler>` tag, and others must appear within a `<metadata>` tag. You can determine the correct hierarchy from the compiler help.

The following is a list of the options returned by `mxmlc -help list advanced`:

```
-benchmark
-compiler.accessible
-compiler.actionscript-file-encoding <string>
-compiler.allow-source-path-overlap
-compiler.as3
-compiler.context-root <context-path>
-compiler.debug
-compiler.defaults-css-files [filename] [...]
-compiler.defaults-css-url <string>
-compiler.define <name> <value>
-compiler.es
-compiler.external-library-path [path-element] [...]
-compiler.fonts.advanced-anti-aliasing
-compiler.fonts.flash-type
-compiler.fonts.languages.language-range <lang> <range>
-compiler.fonts.local-fonts-snapshot <string>
-compiler.fonts.managers [manager-class] [...]
-compiler.fonts.max-cached-fonts <string>
-compiler.fonts.max-glyphs-per-face <string>
-compiler.headless-server
-compiler.include-libraries [library] [...]
-compiler.incremental
-compiler.keep-all-type-selectors
-compiler.keep-as3-metadata [name] [...]
-compiler.keep-generated-actionscript
-compiler.library-path [path-element] [...]
-compiler.locale [locale-element] [...]
-compiler.mxml.compatibility-version <version>
-compiler.namespaces.namespace <uri> <manifest>
-compiler.optimize
-compiler.services <filename>
-compiler.show-actionscript-warnings
-compiler.show-binding-warnings
-compiler.show-shadowed-device-font-warnings
-compiler.show-unused-type-selector-warnings
-compiler.source-path [path-element] [...]
-compiler.strict
-compiler.theme [filename] [...]
-compiler.use-resource-bundle-metadata
-compiler.verbose-stacktraces
-compiler.warn-array-tostring-changes
-compiler.warn-assignment-within-conditional
-compiler.warn-bad-array-cast
-compiler.warn-bad-bool-assignment
-compiler.warn-bad-date-cast
-compiler.warn-bad-es3-type-method
-compiler.warn-bad-es3-type-prop
-compiler.warn-bad-nan-comparison
-compiler.warn-bad-null-assignment
-compiler.warn-bad-null-comparison
-compiler.warn-bad-undefined-comparison
-compiler.warn-boolean-constructor-with-no-args
-compiler.warn-changes-in-resolve
-compiler.warn-class-is-sealed
```

```
-compiler.warn-const-not-initialized
-compiler.warn-constructor-returns-value
-compiler.warn-deprecated-event-handler-error
-compiler.warn-deprecated-function-error
-compiler.warn-deprecated-property-error
-compiler.warn-duplicate-argument-names
-compiler.warn-duplicate-variable-def
-compiler.warn-for-var-in-changes
-compiler.warn-import-hides-class
-compiler.warn-instance-of-changes
-compiler.warn-internal-error
-compiler.warn-level-not-supported
-compiler.warn-missing-namespace-decl
-compiler.warn-negative-uint-literal
-compiler.warn-no-constructor
-compiler.warn-no-explicit-super-call-in-constructor
-compiler.warn-no-type-decl
-compiler.warn-number-from-string-changes
-compiler.warn-scoping-change-in-this
-compiler.warn-slow-text-field-addition
-compiler.warn-unlikely-function-value
-compiler.warn-xml-class-has-changed
-debug-password <string>
-default-background-color <int>
-default-frame-rate <int>
-default-script-limits <max-recursion-depth> <max-execution-time>
-default-size <width> <height>
-dump-config <filename>
-externs [symbol] [...]
-frames.frame [label] [classname] [...]
-help [keyword] [...]
-include-resource-bundles [bundle] [...]
-includes [symbol] [...]
-licenses.license <product> <serial-number>
-link-report <filename>
-load-config <filename>
-load-externs <filename>
-metadata.contributor <name>
-metadata.creator <name>
-metadata.date <text>
-metadata.description <text>
-metadata.language <code>
-metadata.localized-description <text> <lang>
-metadata.localized-title <title> <lang>
-metadata.publisher <name>
-metadata.title <text>
-output <filename>
-raw-metadata <text>
-resource-bundle-list <filename>
-runtime-shared-libraries [url] [...]
-runtime-shared-library-path [path-element] [rsl-url] [policy-file-url]
[rsl-url] [policy-file-url]
-static-link-runtime-shared-libraries
-target-player <version>
-use-network
```

```
-verify-digests
-version
-warnings
```

You'll notice that some of the options you already know, such as `incremental` and `title`, are prefixed (e.g., `compiler.incremental` and `metadata.title`). These prefixed commands are the full commands. The compiler defines aliases that you can use from the command line. That way, the compiler knows when you type **`incremental`**, you really mean `compiler.incremental`. However, when you use a configuration file, you must use the full option names. Prefixes translate to parent nodes. For example, the following sets the `incremental` option to `true` and the `title` option to `Example`:

```
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
  <compiler>
    <incremental>true</incremental>
  </compiler>
  <metadata>
    <title>Example</title>
  </metadata>
</flex-config>
```

In the options list you'll notice that some options are followed by a value enclosed in `<>`. For example, the `title` option is followed by `<text>`. These values indicate that the option value should be a string. For example, as you can see in the preceding sample code, the `<title>` tag has a nested string value of `Example`. If an option is followed by two or more `<value>` values, the option node should contain child tags with the specified names. For example, the `localized-title` option is followed by `<text> <lang>`. Therefore, the following is an example of a configuration file that correctly describes the `localized-title` option:

```
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
  <metadata>
    <localized-title>
      <text>Example</text>
      <lang>en_US</lang>
    </localized-title>
  </metadata>
</flex-config>
```

If an option is followed by `[value] [...]`, it means the option node must contain one or more tags with the name specified. For example, `file-specs` is followed by `[path-element] [...]`. This means that the following is a valid configuration file specifying a `file-specs` value:

```
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
  <file-specs>
    <path-element>Example.mxml</path-element>
  </file-specs>
</flex-config>
```

The following example is also a valid configuration file. This time, it defines several target files to compile.

```
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
  <file-specs>
    <path-element>Example.mxml</path-element>
    <path-element>Example2.mxml</path-element>
    <path-element>Example3.mxml</path-element>
    <path-element>Example4.mxml</path-element>
  </file-specs>
</flex-config>
```

When an option is not followed by anything, it indicates that the value should be Boolean. For example, `incremental` is not followed by anything in the list.

If you would like to get more details on each compiler option, you can review the help documentation provided with Flex as well as issue a help command for a single command. For example, if you want to find what the `-use-network` command is for, you would issue:

```
mxmlc -help use-network
```

Using Ant

Using the compiler from the command line is not the best way to build applications, for the following reasons:

- It's inconvenient because you have to open a command line and type the command each time.
- Because you have to type the command each time, there's a greater chance of introducing errors.
- Not only is opening a command line and typing a command inconvenient, but it's also slow.
- Compiling from the command line doesn't allow you much in the way of features, such as copying and deploying files, testing for dependencies, and so on.

A standard tool used by application developers for scripting application builds is a program called Apache Ant. Ant is an open source tool that runs on Java to automate the build process. This includes testing for dependencies (e.g., the existence of directories), compiling, moving, and copying files, and launching applications. Although you can use *.bat* files or shell scripts to achieve many of Ant's basic tasks, Ant is extremely feature-rich (it offers support for compressing and uncompressing archives, email support, and FTP support, to name just a few) and can better handle potential errors than *.bat* or shell scripts.

If you're not familiar with Ant, the first thing you should do is to download and install Ant from *http://ant.apache.org*. Once you've installed Ant, you should add a new environment variable, called `ANT_HOME`, as well as the Ant *bin* directory to the system path. The `ANT_HOME` environment variable should point to the root directory of the Ant installation on the computer. For example, if Ant is installed at *C:\Ant* on a Windows system, the `ANT_HOME` environment variable should point to *C:\Ant*. Additionally, you

should add the Ant *bin* directory to the system path. For example, if Ant is installed at *C:\Ant*, add *C:\Ant\bin* to the system path.

Ant works by executing a set of tasks, and by default it does not include a task for compiling Flex applications. We could use a task to manually invoke the `mxmlc` compiler, as we would via the command line, but instead Adobe has provided us a set of Flex tasks to simplify Ant use. The tasks allow you to invoke `mxmlc` and `compc` and generate an HTML wrapper. The tasks are included within the SDK distribution in *<sdk dir>\ant*. Installing the provided tasks is easy. You simply copy the *<sdk dir>\ant\lib\flexTasks.jar* file to your *lib* folder within your Ant installation. Once the file is installed, Ant will recognize the new tasks. Ant uses XML files named *build.xml*. The *build.xml* file for a project contains all the instructions that tell Ant how to compile and deploy all the necessary files (e.g., the application). The *build.xml* file consists of a `<project>` root node that contains nested target nodes. The project node allows you to define three attributes:

`name`
: The name of the project

`default`
: The name of the target to run when no other target is specified

`basedir`
: The directory to use for all relative directory calculations

For our sample *build.xml*, the `<project>` node looks like this to start:

```
<?xml version="1.0" encoding="utf-8"?>
<project name="FlexTest" default="compile" basedir="./">
</project>
```

This says that the base directory is the directory in which the file is stored, and the default target is called `compile`. Once the `<project>` root node is set up, you need to instruct Ant to load the Flex tasks and set some basic properties that every Ant build file must contain:

```
<?xml version="1.0" encoding="utf-8"?>
<project name="FlexTest" default="compile" basedir="./">
    <taskdef resource="flexTasks.tasks"
classpath="${basedir}/flexTasks/lib/flexTasks.jar"/>
    <property name="FLEX_HOME" value="C:/flex/sdk"/>
    <property name="APP_ROOT" value="myApp"/>
</project>
```

The basic setup of a build file is now complete. `FLEX_HOME` and `APP_ROOT` are Ant properties that will be useful when configuring tasks later on. As when you declare a variable in code, it is common practice to define properties in Ant in a single location toward the top and to reference them throughout, as we will see in a bit. You can declare any property you wish, but it is common to create at least these two. `FLEX_HOME` should point to the Flex `sdk` root, and `APP_ROOT` should reference your application directory.

Now that we have a basic build file ready, the final step is to set up a `<target>` within the `<project>` node. Multiple `<target>` nodes can exist within a build file, and each target node represents a named collection of tasks. Ant tasks could involve compiling an application, moving files, creating directories, launching applications, creating ZIP archives, using FTP commands, and so on. You can read all about the types of tasks available within Ant at *http://ant.apache.org/manual/tasksoverview.html*. The following defines the `compile` target for our sample *build.xml* file, which makes use of the newly installed task, `mxmlc`:

```
<?xml version="1.0" encoding="utf-8"?>
<project name="FlexTest" default="compile" basedir="./">
    <taskdef resource="flexTasks.tasks"
classpath="${basedir}/flexTasks/lib/flexTasks.jar"/>
    <property name="FLEX_HOME" value="C:/flex/sdk"/>
    <property name="APP_ROOT" value="myApp"/>
    <target name="compile">
        <mxmlc file="${APP_ROOT}/FlexTest.mxm">
        <load-config filename="${FLEX_HOME}/frameworks/flex-config.xml"/>
        <source-path path-element="${FLEX_HOME}/frameworks"/>
    </target>
</project>
```

This `compile` target runs by default because it is set as the default for the project. When you run the Ant build, the `compile` target runs the `mxmlc` task. Nested within the `<mxmlc>` tag you specify the framework and Flex configuration using the `<load-config>` and `<source-path>` nodes. In this build file, we didn't specify that you can place one or more `<arg>` tags that allow you to add arguments to the command. In this case, we're simply adding the `file-specs` option when calling the compiler.

Once you have a valid *build.xml* file you can run it from the command line by running the `ant` command from the same directory as the file:

```
ant
```

This runs the default target in the *build.xml* file located in the same directory. To run a nondefault target, you can specify the target name after the command. To run several targets, specify each target in a list, separated by spaces:

```
ant target1 target2
```

Ant integrates well with most IDEs and is often the preferred choice for build environments. Full coverage of Ant is beyond the scope of this book. For further information on Ant you can review *http://ant.apache.org/*, and for Flex-specific coverage you can review the documentation provided with the Flex SDK.

Compiling Using Flex Builder

If you work with Flex Builder, you can use the built-in options for building. Flex Builder automatically compiles your application for development purposes as you work, but the application produced by this automatic compilation is not suitable for deployment

as it contains debug code that you will want to avoid for deployment unless you intend to debug in a production environment. To compile a production-ready application, use the Export Releaser Build option in the Project→Export release build menu item. By default, the compiled application will be placed in the *bin-release* folder of your Flex project.

Flex Builder runs the application in your default web browser unless you configure it to do otherwise. You can configure what web browser Flex Builder uses by selecting Window→Preferences→General→Web Browser.

Flex Builder builds all projects incrementally by default in debug mode within the *bin-debug* folder. That means it compiles only the elements that have changed since the last build. If you need to recompile all the source code, you need to clean the project, meaning that you instruct the compiler to recompile every necessary class, not just those that have changed since the last compile. You can do that by selecting Project→Clean. This opens the Clean dialog. The Clean dialog has two options: "Clean all projects" and "Clean projects selected below." If you select "Clean projects selected below," it cleans only the projects that you have selected in the list that appears in the dialog. Flex Builder then builds the project or projects the next time it is prompted, either by automatic triggers (saving a file) or when explicitly directed to run a build.

If you want to manually control a build, you must disable the automatic build feature by deselecting Project→Build Automatically. You can then select the Build All, Build Project, or Build Working Set option from the Project menu to manually run a build. The automatic build option is convenient for smaller projects that compile quickly. However, it's frequently helpful to disable automatic build for larger projects that require more time to compile. In such cases, the automatic build feature can cause delays every time you save a file rather than allowing you to build on demand.

Publishing Source Code

Since Flex applications are compiled, the source code for the application is not available by default. This is in contrast with traditional HTML applications in which the user has the option to view the source code from the browser. Although not appropriate for all applications, you do have the option to publish the source code for Flex applications using a Flex Builder feature. When you publish the source code, the user can select a View Source context menu item from Flash Player. The menu option will launch a new browser window that allows the user to view the published source code.

From Flex Builder you can select Project→Export Release Build. The Export Release Build dialog will open, where you can enable the view source. You may also select which source elements you want to publish by clicking on the Choose Source Files button. By default, all project source code and assets are selected. You can also specify the

subdirectory to which to publish the source code files. All the selected ActionScript and MXML files are saved as HTML files.

If the main application entry point is an MXML file, Flex Builder automatically adds the necessary code to enable the View Source context menu item. To manually enable the View Source context menu for an MXML document, you should add the `viewSourceURL` attribute to the `<mx:Application>` tag such that it points to the *index.html* page in the published source code directory.

If you're publishing the source code for an application that uses an ActionScript class as the main entry point, you'll have to enable the context menu item using ActionScript code. This step requires the `com.adobe.viewsource.ViewSource` class. You should then call the static `addMenuItem()` method, passing it a reference to the main class instance and the URL for the source code, like so:

```
ViewSource.addMenuItem(this, "sourcecode/index.html");
```

Deploying Applications

Once you've compiled a Flex application, you next need to deploy the application. Many Flex applications are deployed on the Web, and that will be our focus in this section. If you want to learn more about deploying desktop applications (AIR applications), see Chapter 21.

Every Flex application consists of at least one main *.swf* file. Therefore, at a minimum you will always need to copy at least this one file to the deployment location (typically a web server). However, in addition to the main *.swf*, a Flex application may consist of the following deployable elements:

- An HTML wrapper file
- Data services (web services, Flash Remoting services, etc.)
- Text and XML assets loaded at runtime
- Images loaded at runtime
- Audio and video assets loaded at runtime
- Additional *.swf* files loaded at runtime
- Runtime shared libraries
- Modules

When you deploy an application, you need to make sure that you copy all the necessary files to the deployment locations.

We cover embedding applications in a web browser, along with deeper integration with a web browser, in more detail in Chapter 20.

If you are using Ant, you can easily write a *build.xml* file that copies the necessary files to the deployment directories. Ant natively supports filesystem tasks such as copy and move. It also supports FTP tasks for deploying applications to remote servers.

Summary

This chapter introduced the tool sets and techniques you need to create, configure, compile, and deploy Flex applications to the Web. You learned how to use the command-line compilers and well as how to use build tools such as Apache Ant.

CHAPTER 3
MXML

MXML is a declarative markup language used to create the user interface and to view portions of Flex applications. As the name implies, MXML is an XML-based language. If you're familiar with XML or even HTML, many of the basic MXML concepts we discuss in this chapter will already be familiar to you in a general sense. In this chapter, we'll look at all the basics of working with MXML, including the syntax and structure of the language, the elements of which MXML is composed, creating interactivity in MXML, and how you can use MXML to build applications.

Understanding MXML Syntax and Structure

If you've ever worked with XML or HTML, the structure of MXML will be familiar to you. Even if XML and HTML are unfamiliar to you, you will likely find MXML fairly intuitive. MXML uses tags to create components such as user interface controls (buttons, menus, etc.), and to specify how those components interact with one another and with the rest of the application, including data sources. In the following sections we'll look at how to write MXML code.

Creating MXML Documents

All MXML must appear within MXML documents, which are plain-text documents. You can use any text editor, XML editor, or IDE that can work with text or XML to write MXML, including those listed in the preceding chapter. To create a new MXML document, you can create a new text file with the *.mxml* file extension. If you are using Flex Builder, you can use the program's menus to add either a new MXML application, MXML module, or MXML component. All are MXML documents, differing only in the root element added to the document (as you'll see in this chapter).

XML encoding

Every document can and should have an XML declaration. Many IDEs and XML editors automatically add an XML declaration. Flex Builder adds an XML declaration by

default using UTF-8 as the encoding. You must place the declaration as the first line of code in the MXML document, and unless you have a compelling reason to use a different encoding, you should use UTF-8 for the best compatibility:

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

Note that an XML declaration is not strictly required by the Flex compilers. However, for well-formed MXML, you should always include the XML declaration as it is recommended by the XML 1.0 specification.

Applications, modules, and components

All MXML documents can have just one root node. There are three types of MXML documents, and they are defined by the type of root node they have. The first type of MXML document is an application document. Application documents use `Application` nodes as the root node. All Flex applications must have one application document, and application documents are the only type of document you can compile into a Flex application. The following is an example of a basic application document that Flex Builder creates by default:

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

</mx:Application>
```

Note that the `layout` attribute is not strictly required, but it is shown here because this is the default tag Flex Builder creates.

There are a few items to notice about this example:

- The `Application` node has matching opening and closing tags. The closing tag is prefixed by a forward slash (/). All MXML nodes must be closed as in any well-formed XML.
- The tag name uses an `mx` namespace. You can identify a namespace in a tag because the tag name is prefixed with the namespace identifier followed by a colon. We'll talk more about namespaces in the next section.
- The `Application` tag in this example has two attributes, called `xmlns` and `layout`. You use attributes to set values for a node. In this case, the `xmlns` attribute defines the `mx` namespace prefix (more about this in the next section), and the `layout` attribute defines the way in which the contents of the document will be positioned. The `layout` attribute is optional (we discuss this attribute in more detail in Chapter 6). For now, you can define application documents with an absolute layout or with no explicit layout attribute value. We'll talk more about attributes in "Setting component properties" later in this chapter.

Component documents are used to define *MXML components*, which are encapsulated elements of your application that you can abstract and isolate into their own documents to make your applications more manageable. We'll talk more about custom components in Chapter 9, and Chapter 19. The structure of component documents is similar to that of application documents in all respects except that the root node is not an `Application` tag. Rather, a component document uses an existing component as the root node (which is the superclass for the new MXML document/class). Again, we'll discuss this in much more detail later in this chapter and later in the book. However, for illustrative purposes, here we'll look at a simple example of a component document that is based on a standard Flex framework component called `Canvas`:

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

</mx:Canvas>
```

While the preceding is a complete component document, you can't compile an application from it. All Flex applications require an application document in order to compile, and you can use instances of component documents within the application document. You'll learn more about how to create custom components and use them in a Flex application in Chapter 9.

As you can see in this example, the structure of the document is much the same as that of the application document, but with a difference: the root node is a `Canvas` tag rather than an `Application` tag.

A module document is also remarkably similar to application and component documents, the primary difference being the root tag. A module document is used to define an MXML module, which you'll learn more about in Chapter 9. The root tag for a module document is `Module` as in the following example:

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

</mx:Module>
```

All other MXML code appears within the root node of a document. For example, if you want to add a button to an application, the document might look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
  <mx:Button label="Example Button"></mx:Button>
</mx:Application>
```

Although we haven't yet discussed the button component, you can see quite clearly that the tag that adds the component is nested within the opening and closing tags of the root node. You'll also see that the syntax for the tag that adds the button is similar to that of the `Application` tag. It uses < and > characters to demarcate the tag, and it

uses the same syntax for attributes. The `Button` tag in this example also has an opening and closing tag. If you omitted the closing tag, the compiler would not be able to compile the application. However, in the case of the button component, you would not typically nest any tags within it. Therefore, it is sometimes convenient to be able to open and close a node with just one tag. There is a shortcut to achieve this goal. You can simply add a forward slash immediately prior to the > character of the opening tag. That means you can rewrite the preceding example in the following way:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
    <mx:Button label="Example Button" />
</mx:Application>
```

That covers the fundamentals of MXML structure. We'll be elaborating on how to work with specific components and specialized tags throughout the remainder of the book.

Understanding namespaces

As shown in the preceding section, MXML uses something called a *namespace*. Simply put, a namespace is a unique grouping for elements—in this case Flex libraries. The entire Flex framework is written in ActionScript classes and a few MXML component documents that are stored in external libraries within *.swc* files. These external libraries contain tens if not hundreds of classes (and MXML components). Using these elements from ActionScript is not difficult. However, to use the elements from MXML you have to be able to map the library classes and MXML components to tags. You do this through manifest files and namespaces.

As shown in Chapter 2, a manifest file maps an ActionScript class to an identifier: the MXML tag name. A manifest file in and of itself would be enough to enable access to ActionScript classes and MXML components by way of MXML tags. However, the difficulty is that you need a way to ensure uniqueness of scope for the mappings. For example, the Flex framework defines a mapping called `Button` that points to a class called `mx.controls.Button`—a component that creates a simple user interface button. Yet what if you wanted to create your own class that maps to a `Button` identifier? This poses a problem because you cannot meaningfully have two `Button` identifiers within the same scope. If you did, how would the application know which button you are referencing? This highlights the utility of namespaces.

A namespace allows you to create a unique uniform resource identifier (URI) that corresponds to a particular manifest document. This namespace URI is set when the *.swc* file is compiled, as described in Chapter 2. You may recognize this particular URI from the MXML examples shown in the previous section. Within the MXML document you must tell Flex which namespaces you want the document to use. You can do that using the `xmlns` attribute. If you use the `xmlns` attribute by itself, it defines the default namespace for the document. Therefore, the example that follows on the next page is a valid MXML application document that adds a button component.

```
<?xml version="1.0" encoding="utf-8"?>
<Application xmlns="http://www.adobe.com/2006/mxml" layout="absolute">
  <Button label="Example Button" />
</Application>
```

This example says to use the Flex framework namespace as the default namespace for the document. This means that every tag used in the document is assumed to correspond to one of the mappings in the Flex framework manifest file. Therefore, the `Application` tag maps to the class that corresponds to the `Application` identifier in the manifest file. This is perfectly valid. However, this is not the way in which MXML documents typically utilize the Flex framework namespace; an MXML document may contain tags that shouldn't map to the Flex framework namespace by default. Therefore, it is better not to define that namespace as the default, but rather to use a namespace prefix. By convention we use the `mx` prefix for the Flex framework namespace. You can use a namespace prefix by following `xmlns` with a colon and the prefix before assigning the value, as in the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
  <mx:Button label="Example Button" />
</mx:Application>
```

This example, which is exactly the same as an earlier example, adds the `mx` prefix for the Flex framework namespace. That means you must then prefix all tags that are part of that namespace with the `mx` prefix (e.g., `<mx:Button>`).

By using namespace prefixes, you can create additional namespaces and utilize them within Flex applications. Each namespace can use a different prefix within the MXML document, ensuring that even if two namespaces use the same mapping identifiers, they will not be in conflict. The following example illustrates this:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                xmlns:example="http://www.example.com" layout="absolute">
  <mx:Button label="Example Button" />
  <example:Button />
</mx:Application>
```

This example presupposes that a valid external library is already compiled with the namespace URI of *http://www.example.com* and that the library's manifest file contains a mapping identifier of `Button`. In this example, the application creates one button from the Flex framework and one button from the example library. We'll see more examples of creating custom namespaces for custom libraries in Chapter 20. Although no rule states that you must use the `mx` prefix for the Flex framework namespace, it is the standard convention, and we use that convention in this book.

Components

Flex applications are largely composed of *components*, or modular elements. Technically, a component is an ActionScript class or an MXML component document that

you can instantiate in an application. In some cases the class or component document has been mapped to an identifier via a manifest file, and in some cases you can merely reference the class or component document by way of the fully qualified name. There are many different types of components, but in terms of the Flex framework components, there are two basic categories: visual and non-visual. The visual components consist of the following:

- Containers
- User interface controls

The non-visual components consist of the following:

- Data components
- Utility components

Containers

Containers are types of components that can contain other components. Every application must use containers. At a minimum, the `Application` element itself is a container because you can place other components within it. You use containers for layout. There are containers for vertical layout, horizontal layout, grids, tiles, and all sorts of layout configurations. When you use layout containers, you place other components within them using nested tags. The following uses a `VBox` (a container that automatically arranges the child elements so that they are stacked vertically) to stack two buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
  <mx:VBox>
    <mx:Button label="Example Button 1" />
    <mx:Button label="Example Button 2" />
  </mx:VBox>
</mx:Application>
```

You can nest containers within containers, as the following example shows, by placing an `HBox` (a container that automatically arranges the child elements so that they are placed side by side horizontally) inside a `VBox`:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
  <mx:VBox>
    <mx:Button label="Example Button 1" />
    <mx:Button label="Example Button 2" />
    <mx:HBox>
      <mx:Button label="Example Button 3" />
      <mx:Button label="Example Button 4" />
    </mx:HBox>
  </mx:VBox>
</mx:Application>
```

You can read more about layout containers in Chapter 6.

UI controls

User interface controls are visible interface elements such as buttons, text inputs, lists, and data grids. There are many types of UI controls, and we discuss them in more detail in Chapter 7. You've already had a chance to see several examples with a button control.

Setting component properties

When you work with components, you often need to configure them by setting properties. For example, a `button` component lets you apply a label by setting a property. Every component type has its own unique set of properties that you can set. For example, a `button` and a `VBox` clearly have different properties because they do different things. However, despite the difference in the specific properties available for components, you can set the properties using the same techniques. You can set properties of components in several ways:

- Using tag attributes
- Using nested tags
- Using ActionScript

The simplest and most common way to set properties for a component is to use the tag attributes. We already showed several examples of this technique in earlier code examples. For instance, the `Application` tag allows you to set a `layout` property using a tag attribute, as in the following example:

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

</mx:Application>
```

You'll notice that tag attributes always appear in the opening tag following the tag name. A tag can have many attributes, each separated by spaces. The attributes themselves consist of the attribute name, an equals sign, and the value enclosed in quotation marks.

Almost all components (all visible components) have an `id` property. In most instances of containers and UI controls, you should set the `id` property, because that is how to reference the instance using data binding or ActionScript. The `id` property is the name of the component instance, and it must be unique within the document. The value must also follow a few naming rules. Specifically, the `id` property for a component should consist only of letters, numbers, and underscores, and it should start with either an underscore or a letter, but not a number. The following assigns an `id` value to a button:

```
<mx:Button id="exampleButton" label="Example Button" />
```

You can set most properties (though not the `id` property) using nested tags as an alternative to tag attributes. The nested tags use the same name as the property/attribute, but they must be prefixed with the correct namespace prefix.

The following example assigns a button label using a nested tag:

```
<mx:Button id="exampleButton">
  <mx:label>Example Button</mx:label>
</mx:Button>
```

In most cases, it's preferable to set properties using attributes rather than nested tags because attributes are a more compact and more readable format. However, there are legitimate use cases that justify using nested tags. For example, some properties require complex values that cannot be represented by a string value placed within quotation marks. One such example is the `dataProvider` property for a combo box (a drop-down menu component). The `dataProvider` property of a combo box must be some sort of collection of values. The following example creates a combo box and uses a nested `dataProvider` tag to populate it with values:

```
<mx:ComboBox id="exampleComboBox">
  <mx:dataProvider>
    <mx:ArrayCollection>
      <mx:Array>
        <mx:String>A</mx:String>
        <mx:String>B</mx:String>
        <mx:String>C</mx:String>
        <mx:String>D</mx:String>
      </mx:Array>
    </mx:ArrayCollection>
  </mx:dataProvider>
</mx:ComboBox>
```

Note that you can also set the `dataProvider` property using ActionScript, which would not require nested tags. However, when you want to use MXML to set the `dataProvider` property, you must use nested tags, as in this example.

You can also set properties using ActionScript. When you set an `id` property for a component, you can reference it using that name as an ActionScript object. Most (though not all) component properties have the same names as attributes and as ActionScript properties. We'll look at working with ActionScript in the next chapter.

Non-visual components

As mentioned earlier, there are two types of non-visual components: data components and utility components. *Data components* are used to create data structures, such as arrays and collections, and for making remote procedure calls with protocols like SOAP for web services or AMF for Flash Remoting. You can read more about data components in Chapter 17.

Utility components are components used to achieve functionality. Examples of utility components are those used for creating repeating components and for creating data binding between components. Since utility components are responsible for varied,

generally unrelated tasks, we haven't grouped them all in one chapter. Rather, you'll find discussions of utility components in the context of the topics when you'd most likely use the components. For example, the data binding component is discussed in Chapter 14, and the repeater component is discussed in Chapter 6.

Making MXML Interactive

MXML is useful for creating user interfaces—layout and controls. However, static content is not the hallmark of rich Internet applications. Users expect to be able to interact with Flex applications. There are two basic ways to create interactivity in MXML: handling events and data binding.

Handling Events

Every component does certain things. For example, at a minimum, all visual components can initialize themselves and resize. Most components can do things specific to that component type. For example, a button can respond to a user click. All of these things translate into something called an *event*. An event is a way that a component can notify other parts of the application when some action occurs. When a component sends out this notification, we say that it *dispatches an event*.

The Flex event model is based on the W3C specification. (See *www.w3.org/TR/DOM-Level-3-Events*.)

In Flex all events are dispatched in the form of Event objects. Some events are a more specific type, meaning the event objects are actually instances of a subclass of the Event class. For example, when an Image component loads a file, it dispatches events of type ProgressEvent, which is a subclass of Event. Because all events are of type Event (or a subclass of Event), they all contain the same type of information, including the type of event (i.e., was it a click event or a progress event or an initialize event?) as well as what object dispatched the event. You'll learn more about events and event dispatching details in Chapter 4.

Every type of component has set events that it dispatches. For example, a button component will always dispatch a click event when the user clicks on it (assuming the button is enabled). However, just because a component dispatches an event doesn't mean that anything is receiving a notification. If you want your application to respond to an event, you must tell it to handle the event.

There are several ways you can handle events. One way is to use ActionScript to register listeners. We'll talk about that solution in Chapter 4, when we talk about ActionScript in more detail. In this chapter, we're more interested in the MXML solutions. Within

MXML, you can add inline event handler attributes within a component tag. The event handler attribute name always matches the event name. For example, to handle a click event for a button you use the `click` attribute within the component tag. The value that you assign to an event attribute gets interpreted as ActionScript. The following example handles a button click event and launches an alert window:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
  <mx:Script>
  <![CDATA[
    import mx.controls.Alert;
  ]]>
  </mx:Script>
  <mx:Button id="alertButton" label="Show Alert"
             click="Alert.show('Example')" />
</mx:Application>
```

Even though we haven't yet talked about ActionScript or the `Alert` component, you can see that in this example that the click event attribute is defined to call `Alert.show('Example')`. If you test this example, you'll find that when you click the button, an alert dialog opens with the message that says `Example`.

In this section, our goal was simply to explain the concept of MXML event handling and to show the basic syntax. We'll discuss specific events throughout the book when talking about the components that dispatch the events.

Using Data Binding

Data binding is a feature you can use to link a component to another component or an ActionScript object. Data binding automates changing the value of one object when the value of another object changes. Data binding is an important concept for building Flex applications, and we've dedicated much of Chapter 14 to a detailed discussion of the topic. However, you'll need to understand data binding basics for some of the examples in the intervening chapters.

There are several syntaxes you can employ to enable data binding, but the simplest is a syntax that uses curly braces ({}) to evaluate a statement inline within an MXML tag. In Chapter 14, we'll discuss the additional ways to enable data binding, but before that point, we'll use only the curly brace syntax. The following example uses a text control and a text input control stacked vertically. Each of these controls is a standard Flex framework UI control. The `text` property of each of these controls allows you to read and write the value displayed in the control. In this first example, the text control displays the value `Example`:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
  <mx:VBox>
    <mx:Text id="output" text="Example" width="200" height="200"  />
    <mx:TextInput id="input" />
```