



ADOBE® FLEX® 4.5 Fundamentals

Training from the **Source**

Michael Labriola, Jeff Tapper, and Matthew Boles

Foreword by Adam Lehman, Adobe Flash Builder Product Manager



Adobe® Flex® 4.5 Fundamentals
Training from the Source

Michael Labriola

Jeff Tapper

Matthew Boles

Foreword by Adam Lehman, Adobe Flash Builder Product Manager



Adobe® Flex® 4.5 Fundamentals: Training from the Source

Michael Labriola/Jeff Tapper/Matthew Boles



This Adobe Press book is published by Peachpit.

For information on Adobe Press books, contact:

Peachpit

1249 Eighth Street

Berkeley, CA 94710

510/524-2178

510/524-2221 (fax)

For the latest on Adobe Press books, go to www.adobepress.com

To report errors, please send a note to errata@peachpit.com

Copyright © 2012 by Michael Labriola and Jeffrey Tapper

Adobe Press Editor: Victor Gavenda

Project Editor: Nancy Peterson

Development Editor: Robyn G. Thomas

Technical Editor: Steve Lund

Production Coordinator: Becky Winter

Copy Editor: Jessica Grogan

Compositor: Danielle Foster

Indexer: Emily Glossbrenner

Cover Design: Peachpit Press

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of the book, neither the authors, Adobe Systems, Inc., nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Flash, ColdFusion, and Adobe are registered trademarks of Adobe Systems, Inc. Flex is a trademark of Adobe Systems, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

Printed and bound in the United States of America

ISBN 13: 978-0-321-77712-6

ISBN 10: 0-321-77712-3

9 8 7 6 5 4 3 2 1

*To my wife, Laura, and my daughter, Lilia;
you make life much less quiet, but so much more worthwhile.*

—Michael Labriola

*My efforts on this book are dedicated to my wife, Lisa, and children,
Kaliope and Kagan. Without you to inspire me, this just wouldn't be possible.*

—Jeff Tapper

*To Sandra, my wife, who has made the last 25 years together a joy.
And to Scrappy, my furry fishing buddy.*

—Matthew Boles

Bios

Michael Labriola is a Founding Partner and Senior Consultant at Digital Primates. He has been developing Internet applications since 1995 and has been working with Flex since its 1.0 beta program. Michael is a Flex SDK contributor, architect of both the open source FlexUnit and Spoon Framework projects, and international speaker on Flex and AIR topics who has consulted for many of the world's most recognized brands.

Jeff Tapper is a Founding Partner and Senior Consultant at Digital Primates, a company that provides expert guidance on rich Internet application development and empowers clients through mentoring. He has been developing Internet-based applications since 1995 for a myriad of clients, including Major League Baseball, ESPN, Morgan Stanley, Conde Nast, IBM, Dow Jones, American Express, Verizon, and many others. He has been developing Flex applications since the earliest days of Flex 1. As an instructor, Jeff is certified to teach all of Adobe's courses on Flex, AIR, Flash, and ColdFusion development. He is also a frequent speaker at Adobe Development Conferences and user groups.

Matthew Boles is a Technical Training Specialist for the Adobe Enterprise Training group, and has been developing and teaching courses on Flex since the 1.0 release. Matthew has a diverse background in web development, computer networking, and teaching. He is coauthor of previous versions of this book, as well as a contributing author of the Adobe authorized Flex courseware.

Acknowledgments

Thanks to Robyn, Steve, Jeff, and Matt for their work and dedication to this book. Thanks to my clients and colleagues for the motivation to keep learning new technologies. Thanks to my family for the unwavering support and love. Most importantly, thanks to those who inspire me every day with their words, caring, and wisdom; I promise to always keep trying.

—Michael Labriola

I would like to thank Mike, Matt, Steve, and Robyn for all their hard work, which has helped shape this book. Thanks to Chris Gieger for providing some design love for our application—Chris, sorry we couldn't fully implement your excellent design. Special thanks go to the team at Adobe who has made this all possible. Thanks to the editorial staff at Adobe Press, who was faced with the Herculean task of making our writing intelligible.

—Jeff Tapper

Thanks to Jeff, Mike, and Robyn for making this the easiest book revision I've ever worked on!

—Matthew Boles

Contents

	Forewordx
	Introductionxii
LESSON 1	Understanding Rich Internet Applications	3
	The Evolution of Computer Applications	4
	The Break from Page-Based Architecture	6
	The Advantages of Rich Internet Applications	7
	RIA Technologies	8
	What You Have Learned15
LESSON 2	Getting Started17
	Getting Started with Flex Application Development18
	Creating a Project and an MXML Application18
	Understanding the Flash Builder Workbench24
	Running Your Application28
	Exploring the Flash Builder Debugger34
	Getting Ready for the Next Lessons41
	What You Have Learned43
LESSON 3	Laying Out the Interface45
	Learning About Layouts46
	Laying Out the E-Commerce Application50
	Working with Constraint-Based Layouts58
	Working with View States63
	Refactoring71
	What You Have Learned76
LESSON 4	Using Simple Controls79
	Introducing Simple Controls80
	Displaying Images81
	Building a Detail View85

	Using Data Binding to Link a Data Structure to a Simple Control88
	Using a Form Layout Container to Lay Out Simple Controls.89
	What You Have Learned.92
LESSON 5	Handling Events.95
	Understanding Event Handling96
	Handling System Events107
	What You Have Learned.111
LESSON 6	Using Remote XML Data113
	Using Embedded XML.114
	Using XML Loaded at Runtime119
	Retrieving XML Data via HTTPService.124
	Searching XML with E4X127
	Using Dynamic XML Data133
	Using the XMLListCollection in a Flex Control.137
	What You Have Learned.139
LESSON 7	Creating Classes141
	Building a Custom ActionScript Class142
	Building a Value Object143
	Building a Method to Create an Object.150
	Building Shopping Cart Classes.154
	Manipulating Shopping Cart Data159
	What You Have Learned.167
LESSON 8	Using Data Binding and Collections169
	Examining Data Binding170
	Being the Compiler176
	Understanding Bindable Implications.183
	Using ArrayCollection184
	Refactoring ShoppingCartItem204
	What You Have Learned.205
LESSON 9	Breaking the Application into Components207
	Introducing MXML Components208
	Splitting Off the ShoppingView Component213
	Breaking Out a ProductItem Component221
	Creating Components to Manage Loading the Data230
	What You Have Learned.238

LESSON 10	Using DataGroups and Lists	241
	Using Lists	242
	Using DataGroups	245
	Virtualization with Lists	255
	Displaying Grocery Products Based on Category Selection	257
	What You Have Learned.	259
LESSON 11	Creating and Dispatching Events.	261
	Understanding the Benefits of Loose Coupling.	262
	Dispatching Events.	263
	Declaring Events for a Component.	267
	Identifying the Need for Custom Event Classes.	269
	Building and Using the UserAcknowledgeEvent	270
	Understanding Event Flow and Event Bubbling	274
	Creating and Using the ProductEvent Class	280
	What You Have Learned.	289
LESSON 12	Using the Flex DataGrid	291
	Introducing DataGrids and Item Renderers	292
	Displaying the ShoppingCart with a DataGrid	292
	What You Have Learned.	309
LESSON 13	Using Drag and Drop	311
	Introducing the Drag and Drop Manager	312
	Enhanced Dragging and Dropping Between Two Lists	313
	Standard Dragging and Dropping Between a DataGrid and a List	315
	Using a Non-Drag-Enabled Component in a Drag-and-Drop Operation	321
	Dragging a Grocery Item to the Shopping Cart.	326
	What You Have Learned.	331
LESSON 14	Implementing the Checkout Process	333
	Introducing Navigation with States	334
	Introducing Two-Way Bindings	334
	Creating the OrderInfo valueObject	335
	Creating CheckoutView.	337
	Creating CreditCardInfo.	345
	Creating Review	350
	Completing the Order.	355
	What You Have Learned.	358

LESSON 15	Using Formatters and Validators	361
	Introducing Formatters and Validators	362
	Using Formatter Classes	364
	Examining a Second Locale	368
	Using Validator Classes	369
	What You Have Learned.	372
LESSON 16	Customizing a Flex Application with Styles	375
	Applying a Design with Styles and Skins	376
	Cleaning Up the Appearance	376
	Applying Styles	377
	Changing CSS at Runtime	395
	What You Have Learned.	397
LESSON 17	Customizing a Flex Application with Skins	399
	Understanding the Role of Skins in a Spark Component	400
	Understanding the Relationship between Skins and States	404
	Creating a Skin for the Application.	413
	What You Have Learned.	417
LESSON 18	Creating Custom ActionScript Components	419
	Introducing Components with ActionScript 3.0	420
	Building Components Can Be Complex.	420
	Understanding Flex Components	421
	Why Make Components?.	422
	Defining a Component	424
	Creating the Visuals	432
	Adding Functionality to the Component.	439
	Creating a Renderer for the Skin	450
	What You Have Learned.	452
APPENDIX	Setup Instructions	455
	Software Installation.	455
	Importing Projects	458
	Index	462

Foreword

Over a decade ago, Adobe (then Macromedia) coined the term rich Internet application, or RIA, to describe the future of browser-based applications. This new breed of application supplemented existing server-based applications with an enhanced client-side user experience. As Internet users became increasingly sophisticated, demand for improved user experiences grew. At the center of this paradigm shift was Adobe Flex, a simple and light-weight framework for developing applications.

Once a novelty, Internet usage on phones and tablets has exploded. Users can now access the Internet more from mobile devices than from personal computers. As such, user demand for browser-based applications is shifting to applications installed on devices. Yet again, the Flex framework can be found leading the charge. With the release of the Flex 4.5 SDK, Flex applications can now be deployed as native applications to Android, Apple iOS, and Blackberry devices. With this book, you hold in your hands all the knowledge and best practices necessary to deliver killer applications for not just one of the leading mobile platforms...but all of them!

Adobe Flex is composed of a number of elements. It uses a declarative markup language called MXML to help structure your application and ActionScript, a highly productive scripting language, to glue all the pieces together. The framework also has built-in support for CSS and a simple but comprehensive skinning model. These complimentary languages will probably look familiar to those with HTML and JavaScript experience. In addition to the languages that power Flex, the framework provides layout containers, form controls, validators, effects, state management frameworks, a multipurpose animation library, and much more to help you rapidly build the next generation of web applications.

Of course, what good is a slick interface if you can't connect it to live data and services? Fortunately, Flex offers a multitude of ways to connect to nearly any backend service, whether it is raw XML over HTTP, SOAP web services, or the blazingly fast remoting protocol called Action Message Format (AMF). If you're looking for an enterprise-grade data management solution to share data with multiple users simultaneously, Flex offers tight integration with the Adobe Digital Enterprise Platform and Adobe LiveCycle DataServices.

Most of the improvements in Flex 4.5 are focused around mobile and device development. Rather than introducing a separate mobile version of Flex, we upgraded the existing framework for mobile development. You can now use the same tools and languages to build a Flex mobile application that you do to build a Flex application for the browser of the desktop. Built on the foundation of Spark, the next generation component model introduced in Flex 4, Flex 4.5 continues to add new components and capabilities. The Flex compiler has also undergone numerous improvements to ensure applications run faster with even less memory.

Flex is open source and free. Outside this book, you don't have to purchase anything else to develop rich Internet applications for the browser, desktop, or mobile devices. You can just open your favorite text editor, write some code, and compile your application at the command line. But if you're like me, you'll probably want some better tooling support. This book uses Adobe Flash Builder 4.5, the premiere IDE for Flex and ActionScript development. Flash Builder 4.5's rock-solid code editor and intuitive features, like Quick Assist, will make you fall in love with ActionScript coding. If that isn't enough, Flash Builder 4.5 supports the new mobile workflow, from the creation of a new mobile project to debugging your application live on a connected device. Additionally, there is a large and vast ecosystem of third-party tools, libraries, and extensions (some written by your authors!) to enhance productivity and aid in the development of your applications.

There is a wealth of reference information on Flex freely available on the Internet, but to build the next killer app, you need to know how to put all the pieces together. *Adobe Flex 4.5: Training from the Source* draws from the expertise of its authors to present lessons that not only introduce you to the Flex framework but also teach you the best practices you need to be successful.

Times are changing. Whether its browser, desktop, or mobile devices, the Flex SDK and Adobe Flash Builder provides the tools you need to build a better Internet. The next fabulous app is just a few clicks away.

Adam Lehman
Senior Product Manager
Adobe Systems, Inc.

Introduction

Macromedia introduced Flex in 2004 so that developers could write web applications for the nearly ubiquitous Flash platform. These applications benefited from the improved design, usability, and portability that Flex made possible, dramatically changing the user experience. These features are a cornerstone of Web 2.0, a new generation of Internet applications focused on creativity and collaboration.

Since the introduction of Flex, Macromedia—and now Adobe—has released versions 1.5, 2, 3, 4, and 4.5 of Flex. With each subsequent version, creating rich, compelling, intuitive applications has gotten easier, and the bar has been raised on users' expectations of web applications. Countless organizations have discovered the benefits of Flex and have built and deployed applications that run on the Flash platform.

But Flex 1 and 1.5 were most definitely not mass-market products. The pricing, lack of IDE, limited deployment options, and other factors meant that those early versions of Flex were targeted specifically for large and complex applications as well as for sophisticated developers and development. However, with the new releases of the Flex product line, all this has changed.

Flex 2 was released in 2006 and made Flex development a possibility for many more people, as it included a free software development kit (SDK). With the open sourcing of Flex 3, and the announcement of free versions of Flash Builder for students, Flex development is within the grasp of any developer with enough foresight to reach for it. The release of Flex 4 made it even easier to build rich, efficient, cutting-edge applications, and streamlined the workflow between designer and developer, greatly easing the process of bringing intuitive, compelling designs to even more Flex applications. In this latest release, Flex 4.5, Adobe has further extended the reach of Flex, making it possible to deploy applications not only to browsers and desktops, but to phones, tablets, televisions, and other connected devices.

Getting started with Flex is easy. Flex itself is composed of two languages: MXML, an XML-based markup language, and ActionScript, the language of Flash Player. MXML tags are easy to learn (especially when Flash Builder writes them for you). ActionScript has a steeper learning curve, but developers with prior programming and scripting experience will pick it up easily. Still, there's more to Flex development than MXML and ActionScript.

To be a successful Flex developer, you'll need to understand a number of concepts, including the following:

- How Flex applications should be built (and how they should not)
- What the relationships between MXML and ActionScript are, and when to use each
- How to load data into a Flex application
- How to use the Flex components, and how to write your own
- What the performance implications are of the code you write
- Which practices you should employ to write code that is scalable, manageable, and reusable

Developing these skills is where this book comes in. As the authors, we have distilled our hard-earned Flex expertise into a series of lessons that will jump-start your own Flex development. Starting with the basics, and then incrementally introducing additional functionality and know-how, the author team guides your journey into the exciting world of RIAs, ensuring success every step of the way.

Flex is powerful, highly capable, fun, and incredibly addictive. And *Adobe Flex 4.5: Training from the Source* is the ideal tour guide on your journey to the next generation of application development.

Adobe Flex 4.5: Training from the Source is an update to the popular *Adobe Flex 4: Training from the Source*. It is our sincere intention that readers of the earlier book, as well those who are first exploring Flex with this book, will find this content compelling. Since the release of our previous book, the Flex SDK has been improved, with features that include:

- Support for internationalization of Flex applications
- Additional components, such as the DataGrid, added to the Spark component set
- Support for deploying applications to desktops, browsers, phones, tablets, and other connected devices
- And much more

It's an incredible time to be an RIA developer, and we hope that this book provides you with all the tools you need to get started with Flex.

Prerequisites

To make the most of this book, you should at the very least understand web terminology. This book isn't designed to teach you anything more than Flex, so the better your understanding of the World Wide Web, the better off you'll be. This book is written assuming that you're comfortable working with programming languages and that you're working with a server-side language such as Java, .NET, PHP, or ColdFusion. Although knowledge of server-side technologies is not required to succeed with this book, we invoke many comparisons and analogies to server-side web programming. This book is not intended as an introduction to programming or as an introduction to object-oriented programming (OOP). Experience with OOP is not required, although if you have no programming experience at all, you might find the materials too advanced.

Outline

As you'll soon discover, this book mirrors real-world practices as much as possible. Where certain sections of the book depart from what would be considered a real-world practice, every attempt has been made to inform you. The exercises are designed to get you using the tools and the interface quickly so that you can begin to work on projects of your own with as smooth a transition as possible.

This curriculum should take approximately 28–35 hours to complete and includes the following lessons:

Lesson 1: Understanding Rich Internet Applications

Lesson 2: Getting Started

Lesson 3: Laying Out the Interface

Lesson 4: Using Simple Controls

Lesson 5: Handling Events

Lesson 6: Using Remote XML Data

Lesson 7: Creating Classes

Lesson 8: Using Data Binding and Collections

Lesson 9: Breaking the Application into Components

Lesson 10: Using DataGroups and Lists

Lesson 11: Creating and Dispatching Events

Lesson 12: Using the Flex DataGrid

Lesson 13: Using Drag and Drop

Lesson 14: Implementing the Checkout Process

Lesson 15: Using Formatters and Validators

Lesson 16: Customizing a Flex Application with Styles

Lesson 17: Customizing a Flex Application with Skins

Lesson 18: Creating Custom ActionScript Components

Who Is This Book For?

All the content of this book should work well for users of Flash Builder on any of its supported platforms. The earlier “Prerequisites” section details what a reader should know prior to reading this, in order to get the most out of this book.

The Project Application

Adobe Flex 4.5: Training from the Source includes many comprehensive tutorials designed to show you how to create a complete application using Flex. The application that you’ll create is an online grocery store that displays data and images and takes a user through the checkout process, ending just before the data would be submitted to a server.

By the end of the book, you’ll have built the entire application using Flex. You’ll begin by learning the fundamentals of Flex and understanding how you can use Flash Builder in developing the application. In the early lessons, you’ll use Design mode to begin laying out the application, but as you progress through the book and become more comfortable with the languages used by Flex, you’ll spend more and more time working in Source mode, which gives you the full freedom and flexibility of directly working with code. By the end of the book, you should be fully comfortable working with the Flex languages and may even be able to work without Flash Builder by using the open source Flex SDK and its command-line compiler.

Errata

Although we have made every effort to create a flawless application and book, occasionally we or our readers find problems. The errata for the book will be posted at www.flexgrocer.com.

Standard Elements in the Book

Each lesson in this book begins by outlining the major focus of the lesson at hand and introducing new features. Learning objectives and the approximate time needed to complete all the exercises are also listed at the beginning of each lesson. The projects are divided into exercises that demonstrate the importance of each skill. Every lesson builds on the concepts and techniques learned in the previous lessons.

The following are some consistent elements and styles you'll encounter throughout the book:

▶ **TIP:** An alternative way to perform a task or a suggestion to consider when applying the skills you are learning.

* **NOTE:** Additional background information to expand your knowledge, or advanced techniques you can explore to further develop your skills.

▼ **CAUTION!** Information warning you of a situation you might encounter that could cause errors, problems, or unexpected results.

Boldface text: Words that appear in **boldface** are terms that you must type while working through the steps in the lessons.

Boldface code: Lines of code that appear in **boldface** within code blocks help you easily identify changes in the block to be made in a specific exercise step.

```
<mx:HorizontalList dataProvider="{dp}"
  labelFunction="multiDisplay"
  columnWidth="130"
  width="850"/>
```

Code in text: Code or keywords appear slightly different from the rest of the text so you can identify them easily.

Code block: To help you easily identify ActionScript, XML, and HTML code within the book, the code has been styled in a special font that's different from the rest of the text. Single lines of ActionScript code that are longer than the margins of the page are wrapped to the next line. They are designated by an arrow at the beginning of the continuation of a broken line and are indented under the line from which they continue. For example:

```
public function Product (_catID:Number, _prodName:String,  
    ➤ _unitID:Number, _cost:Number, _listPrice:Number,  
    ➤ _description:String, _isOrganic:Boolean, _isLowFat:Boolean,  
    ➤ _imageName:String)
```

Italicized text: *Italics* are used to show *emphasis* or to introduce *new vocabulary*.

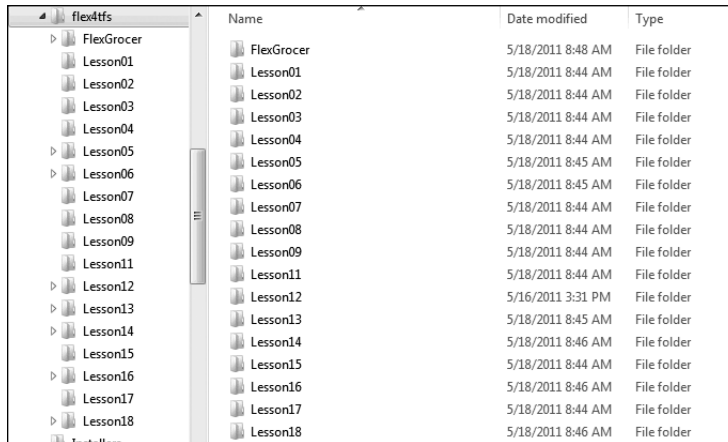
Italics are also used for placeholders, which indicate that a name or entry may change depending on your situation. For example, in the path *driveroot:/flex4tfs/flexgrocer*, you would substitute the actual name of your root drive for the placeholder.

Menu commands and keyboard shortcuts: There are often multiple ways to perform the same task in Flash Builder. The different options will be pointed out in each lesson. Menu commands are shown with angle brackets between the menu names and commands: Menu > Command > Subcommand. Keyboard shortcuts are shown with a plus sign between the names of keys to indicate that you should press the keys simultaneously; for example, Shift+Tab means that you should press the Shift and Tab keys at the same time.

CD-ROM: The CD-ROM included with this book includes all the media files, starting files, and completed projects for each lesson in the book. These files are located in the start and complete directories. Lesson 1, “Understanding Rich Internet Applications,” does not include exercises. If you need to return to the original source material at any point, you can restore the FlexGrocer project. Some lessons include an intermediate directory that contains files in various stages of development in the lesson. Other lessons may include an independent directory that is used for small projects intended to illustrate a specific point or exercise without impacting the FlexGrocer project directly.

Anytime you want to reference one of the files being built in a lesson to verify that you are correctly executing the steps in the exercises, you will find the files organized on the CD-ROM under the corresponding lesson. For example, the files for Lesson 4 are located on the CD-ROM in the Lesson04 folder, in a project named FlexGrocer.fxp.

The directory structure of the lessons you'll be working with is as follows:



Name	Date modified	Type
FlexGrocer	5/18/2011 8:48 AM	File folder
Lesson01	5/18/2011 8:44 AM	File folder
Lesson02	5/18/2011 8:44 AM	File folder
Lesson03	5/18/2011 8:44 AM	File folder
Lesson04	5/18/2011 8:44 AM	File folder
Lesson05	5/18/2011 8:45 AM	File folder
Lesson06	5/18/2011 8:45 AM	File folder
Lesson07	5/18/2011 8:44 AM	File folder
Lesson08	5/18/2011 8:44 AM	File folder
Lesson09	5/18/2011 8:44 AM	File folder
Lesson11	5/18/2011 8:44 AM	File folder
Lesson12	5/16/2011 3:31 PM	File folder
Lesson13	5/18/2011 8:45 AM	File folder
Lesson14	5/18/2011 8:46 AM	File folder
Lesson15	5/18/2011 8:44 AM	File folder
Lesson16	5/18/2011 8:46 AM	File folder
Lesson17	5/18/2011 8:44 AM	File folder
Lesson18	5/18/2011 8:46 AM	File folder

Directory structure

Adobe Training from the Source

The *Adobe Training from the Source* and *Adobe Advanced Training from the Source* series are developed in association with Adobe and reviewed by the product support teams. Ideal for active learners, the books in the *Training from the Source* series offer hands-on instruction designed to provide you with a solid grounding in the program's fundamentals. If you learn best by doing, this is the series for you. Each *Training from the Source* title contains hours of instruction on Adobe software products. They are designed to teach the techniques that you need to create sophisticated professional-level projects. Each book includes a CD-ROM that contains all the files used in the lessons, completed projects for comparison, and more.

What You Will Learn

You will develop the skills you need to create and maintain your own Flex applications as you work through these lessons.

By the end of the book, you will be able to:

- Use Flash Builder to build Flex applications.
- Understand MXML, ActionScript 3.0, and the interactions of the two.
- Work with complex sets of data.

- Load data using XML.
- Handle events to allow interactivity in an application.
- Create your own event classes.
- Create your own components, either in MXML or ActionScript 3.0.
- Apply styles and skins to customize the look and feel of an application.
- And much more.

Minimum System Requirements

Windows

- 2 GHz or faster processor
- 1 GB of RAM (2 GB recommended)
- Microsoft Windows XP with Service Pack 3, Windows Vista Ultimate or Enterprise (32 or 64 bit running in 32-bit mode), Windows Server 2008 (32 bit), or Windows 7 (32 or 64 bit running in 32-bit mode)
- 1 GB of available hard-disk space
- Java Virtual Machine (32 bit): IBM JRE 1.6, or Sun JRE 1.6
- 1024x768 display (1280x800 recommended) with 16-bit video card
- Flash Player 10.2 or later

Macintosh

- Intel processor based Mac
- OS X 10.6 (Snow Leopard)
- 1 GB of RAM (2 GB recommended)
- 1.5 GB of available hard-disk space
- Java Virtual Machine (32 bit): JRE 1.6
- 1024x768 display (1280x800 recommended) with 16-bit video card
- Flash Player 10.2 or later

The Flex line of products is extremely exciting, and we're waiting to be amazed by what you will do with it. With a strong foundation in Flex, you can expand your set of skills quickly.

Flex is not difficult to use for anyone with programming experience. With a little bit of initiative and effort, you can fly through the following lessons and be building your own custom applications and sites in no time.

Additional Resources

Flex Community Help

Flex Community Help brings together active Flex users, Adobe product team members, authors, and experts to give you the most useful, relevant, and up-to-date information about Flex. Whether you're looking for a code sample, an answer to a problem or question about the software, or want to share a useful tip or recipe, you'll benefit from Community Help. Search results will show you not only content from Adobe, but also from the community.

With Adobe Community Help you can:

- Fine-tune your search results with filters that let you narrow your results to just Adobe content, community content, just the ActionScript Language Reference, or even code samples.
- Download core Adobe Help and ActionScript Language Reference content for offline viewing via the new Community Help AIR application.
- See what the community thinks is the best, most valuable content via ratings and comments.
- Share your expertise with others and find out what experts have to say about using your favorite Adobe products.

If you have installed Flash Builder 4.5 or any Adobe CS5 product, then you already have the Community Help application. This companion application lets you search and browse Adobe and community content, plus you can comment and rate any article just like you would in the browser. However, you can also download Adobe Help and reference content for use offline. You can also subscribe to new content updates (which can be downloaded automatically) so that you'll always have the most up-to-date content for your Adobe product at all times. You can download the application from <http://www.adobe.com/support/chc/index.html>.

Community Participation

Adobe content is updated based on community feedback and contributions: You can contribute content to Community Help in several ways: add comments to content or forums, including links to web content; publish your own content via the Community Publishing System; or contribute Cookbook Recipes. Find out how to contribute at www.adobe.com/community/publishing/download.html.

Community Moderation and Rewards

More than 150 community experts moderate comments and reward other users for helpful contributions. Contributors get points: 5 points for small stuff like finding typos or awkward wording, up to 200 points for more significant contributions like long tutorials, examples, cookbook recipes, or Developer Center articles. A user's cumulative points are posted to their Adobe profile page and top contributors are called out on leader boards on the Help and Support pages, Cookbooks, and Forums. Find out more at www.adobe.com/community/publishing/community_help.html.

Frequently Asked Questions

You might find the following resources helpful for providing additional instruction:

For answers to frequently asked questions about Community Help see <http://community.adobe.com/help/profile/faq.html>.

Adobe Flex and Flash Builder Help and Support www.adobe.com/support/flex/ is where you can find and browse Help and Support content on adobe.com.

Adobe TV <http://tv.adobe.com> is an online video resource for expert instruction and inspiration about Adobe products, including a How To channel to get you started with your product.

Adobe Developer Connection www.adobe.com/devnet is your source for technical articles, code samples, and how-to videos that cover Adobe developer products and technologies.

Cookbooks <http://cookbooks.adobe.com/home> is where you can find and share code recipes for Flex, ActionScript, AIR, and other developer products.

Resources for educators www.adobe.com/education includes three free curriculums that use an integrated approach to teaching Adobe software and can be used to prepare for the Adobe Certified Associate exams.

Also check out these useful links:

Adobe Forums <http://forums.adobe.com> lets you tap into peer-to-peer discussions, questions, and answers on Adobe products.

Adobe Marketplace & Exchange www.adobe.com/cfusion/exchange is a central resource for finding tools, services, extensions, code samples, and more to supplement and extend your Adobe products.

Adobe Flex product home page www.adobe.com/products/flex is the official home page from Adobe for Flex related products.

Adobe Labs <http://labs.adobe.com> gives you access to early builds of cutting-edge technology, as well as forums where you can interact with both the Adobe development teams building that technology and other like-minded members of the community.

Adobe Certification

The Adobe Certified program is designed to help Adobe customers and trainers improve and promote their product-proficiency skills. There are four levels of certification:

- Adobe Certified Associate (ACA)
- Adobe Certified Expert (ACE)
- Adobe Certified Instructor (ACI)
- Adobe Authorized Training Center (AATC)

The Adobe Certified Associate (ACA) credential certifies that individuals have the entry-level skills to plan, design, build, and maintain effective communications using different forms of digital media.

The Adobe Certified Expert (ACE) program is a way for expert users to upgrade their credentials. You can use Adobe certification as a catalyst for getting a raise, finding a job, or promoting your expertise.

If you are an ACE-level instructor, the Adobe Certified Instructor (ACI) program takes your skills to the next level and gives you access to a wide range of Adobe resources.

Adobe Authorized Training Centers offer instructor-led courses and training on Adobe products, employing only Adobe Certified Instructors. A directory of AATCs is available at <http://partners.adobe.com>.

For information on the Adobe Certified program, visit www.adobe.com/support/certification/main.html.

This page intentionally left blank

LESSON 1

What You Will Learn

In this lesson, you will:

- Explore alternatives to page-based architecture
- See the benefits of rich Internet applications (RIAs)
- Compare RIA technologies

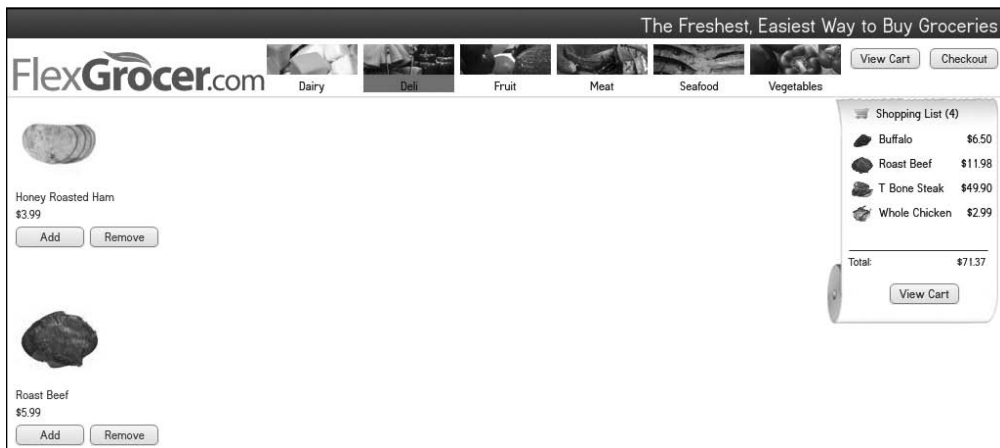
Approximate Time

This lesson takes approximately 30 minutes to complete.

LESSON 1

Understanding Rich Internet Applications

Computers have played a role in business environments for more than five decades. Throughout that time, the roles of client and server have constantly evolved. As businesses and their employees have become more comfortable delegating responsibilities to computers, the look, feel, and architecture of computerized business applications have changed to meet their demands. Today businesses are asking for even faster, lighter, and richer Internet applications. In this lesson, you will learn about this evolutionary environment and understand the business requirements that push developers to build rich Internet applications (RIAs).



You will use Flex to build the FlexGrocer application seen here.

The Evolution of Computer Applications

In the earliest days of computerized business applications, all the processing took place on mainframes, with the client having no role other than displaying information from the server and accepting user input. This setup was largely dictated by the high cost of processing power. Spreading powerful clients throughout the enterprise was simply not affordable, so all processing was consolidated and “dumb terminals” provided the user interaction.

As memory and processing power became cheaper, dumb terminals were replaced by microcomputers (or personal computers). With the added power, more desktop applications, such as word processors and spreadsheets, could run as stand-alone applications, so no server was necessary. One challenge faced by organizations with microcomputers was a lack of centralized data. While the mainframe era had everything centralized, the age of microcomputer-distributed data made it more difficult to centralize business rules and synchronize data across the enterprise.

To help resolve this issue, several vendors released platforms that sought to combine the strengths of the microcomputer with those of the mainframe, which led to the birth of client/server systems. These platforms afforded end users the power and ease of microcomputers while allowing business logic and data to be stored in, and accessed from, a centralized location. The new challenge introduced with client/server systems was distribution. Anytime changes needed to be made to client applications, IT departments had to manually reinstall or upgrade the software on every single desktop computer that ran the software. Large companies found they needed a full-time IT staff whose primary responsibility was keeping the software on the end users' desktops current.

With the explosive growth of the Internet in the 1990s, a new model for business applications became available. In this model, a web browser acted as a thin client whose primary job was to render HTML (Hypertext Markup Language) and to send requests back to an application server that dynamically composed and delivered pages to the client. This is often referred to as a “page-based architecture.” This model successfully solved the distribution problem of the client/server days; the specific page of the application was downloaded from the server each time an end user needed it, so updates could be made in a single centralized place and automatically distributed to the entire user base. This model was and continues to be successful for many applications; however, it also creates significant drawbacks and limitations. In reality, these early Internet applications bore a great resemblance to mainframe applications, in that all the processing was centralized at the server, and the client only rendered data and captured user feedback. The biggest problems with this model surrounded the user interface (UI). Many of the conveniences that end users gained in the client/server era were lost, and the UI was limited by the capabilities of HTML. For example, desktop software as well as client/

server applications frequently allows drag-and-drop. However, HTML applications almost never do; they are prevented by the complexities of—and a lack of cross-browser support for—the DHTML (Dynamic HTML) elements that are required to implement drag-and-drop in a pure HTML/DHTML solution.

In most cases, the overall sophistication of the client-side solutions that could be built and delivered was greatly reduced. Although the web has offered great improvements in the ease and speed of deploying applications, the capabilities of web-based business applications took a big step backward because browser-based applications had to adapt to the limitations of the web architecture: HTML and the stateless nature of Hypertext Transport Protocol (HTTP).

Today, the demands on Internet-based applications continue to grow and are often quite different from the demands of their mid-1990s counterparts. End users and businesses are demanding more from their investments in Internet technology. Many companies are looking toward richer models for Internet applications—models that combine the media-rich power of the traditional desktop with the ease of deployment and content-rich nature of web applications.

As Internet applications begin to be used for core business functions, the maintainability of those applications becomes more crucial. The maintainability of an application is directly related to the application's architecture. Sadly, many web applications were built with little thought about the principles of application architecture and are therefore difficult to maintain and extend. Today, it is easier to build a solid architecture for an application by providing a clean separation between the business, data access, and presentation areas. With the introduction of elements such as web services, the concept of a service-oriented architecture (SOA) has become more feasible for web-based applications.

Rich Internet applications are the next step in this ongoing evolution. They exist to meld the usability and functionality of the client/server era with the ease of distribution provided by the Internet era. To meet the demands of businesses, RIAs should be able to do the following:

- Provide an efficient, high-performance way to execute code, content, and communications. In the next section of this lesson, you'll explore the limitations of the standard HTML-based applications and learn that traditional page-based architectures have a number of performance-related challenges.
- Provide powerful and extensible object models to facilitate interactivity. Web browsers have progressed in recent years in their capability to support interactivity through the Document Object Model (DOM) via JavaScript and DHTML (including HTML 5), but they still lack standardized cross-platform and cross-browser support. Although there are drafts of standards that may address these limitations in the future, building RIAs with these tools today so that they work on a variety of browsers and operating systems involves creating multiple versions of the same application.

- Enable using server-side objects via Web Services or similar technologies. The promise of RIAs includes the capability to cleanly separate presentation logic and user interfaces from the application logic housed on the server.
- Allow for deployment to a variety of devices. Applications are no longer used just from computers, they are also accessed from phones, televisions, tablets, and other connected devices.
- Enable use of Internet applications when offline. As devices continue to grow in popularity, one of the serious limitations of Internet applications is the requirement that the machine running the application remains connected to the Internet. Although users can be online the vast majority of the time, business travelers know there are times when an Internet connection is not possible. A successful RIA should enable users to be productive with or without an active connection.

The Break from Page-Based Architecture

For experienced web developers, one of the biggest challenges in building RIAs is breaking away from a page-based architecture. Traditional web applications are centered on the concept of a web page. Regardless of which server-side technologies (if any) are used, the flow goes something like this:

1. The user opens a browser and requests a page from a web server.
2. The web server receives the request.
3. The web server hands the request to an application server to dynamically assemble the web page, or it retrieves a static page from the file system.
4. The web server sends the page (dynamic or static) back to the browser.
5. The browser draws the page in place of whatever was previously displayed.

Even when most of the content of the previous page is identical to the new page, the entire new page needs to be sent to the browser and rendered. This is one of the inefficiencies of traditional web applications: Each user interaction requires a new page to be loaded in the browser. One of the key goals of RIAs is to reduce the amount of extra data transmitted with each request. Rather than download an entire page, why not download only the data that has changed and update the page the user is viewing? This is the way standard desktop and client/server applications work.

Although this goal seems simple and is readily accepted by developers taking their first plunge into RIA development, web developers often bring a page-based mindset to RIAs and struggle to understand how to face the challenges from the page-based world, such as “maintaining state.” For example, after users log in, how do you know who they are and what they are allowed to do as they navigate around the application?

Maintaining state was a challenge introduced by web-based applications. HTTP was designed as a stateless protocol: Each request to the server was an atomic unit that knew nothing about previous requests. This stateless nature allowed for greater efficiency and redundancy because a connection did not need to be held open between browser and server. Each new page request lasted only as long as the server spent retrieving and sending the page, allowing a single server to handle a much larger number of simultaneous requests.

But the stateless nature of the web added challenges for application developers. Usually, applications need to remember information about the user: login permissions, items added to a shopping cart, and so on. Without the capability to remember this data from one request to the next, true application development would not be possible. To help solve this problem, a series of solutions were implemented revolving around a unique token being sent back to the server with each request (often as cookies, which are small text files containing application-specific identifiers for an individual user) and having the server store the user’s information.

Unlike traditional web applications, RIAs can bypass many of these problems. Because the application remains in client RAM the entire time it’s used (instead of being loaded and unloaded like a page-based model), variables can be set once and accessed throughout the application’s life cycle.

A different approach to handling state is just one of many places in which building applications requires a slightly different mindset than web application development. In reality, web-based RIAs bear more resemblance to client/server applications than they do to web applications.

The Advantages of Rich Internet Applications

Unlike the dot-com boom days of the mid- to late 1990s, businesses are no longer investing in Internet technologies simply because they are “cool.” To succeed, a new technology must demonstrate real return on investment and truly add value. RIAs achieve this on several levels: They can reduce development costs and add value throughout an organization.

Business Managers

RIAs make it easier for end users to work with the software, which leads to an increase in the number of successful transactions. This increase can be quantified: Businesses can measure things like the productivity of employees using intranet applications or the percentage of online shoppers who complete a purchase. More productive employees, making fewer mistakes, can drastically reduce labor costs; and growing online sales can increase revenue and decrease opportunities lost to competitors.

IT Organizations

Breaking away from page-based architectures reduces the load on web servers and reduces overall network traffic. Rather than entire pages being transmitted over and over again, an entire application is downloaded once; then, the only communication back and forth with the server is the data to be presented on the page. Easing the server load and network traffic can noticeably reduce infrastructure costs. RIAs that are developed using sound architectural principles and best practices can also greatly increase the maintainability of an application as well as greatly reduce the development time to build the application.

End Users

A well-designed RIA greatly reduces users' frustration levels because they no longer need to navigate several pages to find what they need, nor do they have to wait for a new page to load before continuing to be productive. Additionally, the time users spend learning how to use an application can be greatly reduced. As the success of devices such as Apple's iPhone and Google's Android-based phones clearly demonstrates, a rich user experience matters, and people are willing to pay for it. End user experiences are one of the greatest benefits of RIAs. Today, there are a number of excellent applications that would not be possible without the concepts of an RIA, such as the MLB.TV Media Player for Major League Baseball, which delivers real-time and on-demand games with statistics and commentary, or the FedEx Custom Critical shipment tracking dashboard that tracks high-value shipments as they move toward delivery. These applications provide excellent examples of the ease of use an RIA can offer an end user.

RIA Technologies

Today, developers have several technology choices when they start building RIAs. Among the more popular technologies are those based on HTML, such as AJAX (Asynchronous JavaScript and XML) and HTML 5, and those based on plug-ins. Plug-in-based options

execute code written in languages such as MXML/ActionScript or XAML/.NET in a virtual machine (Flash Player and Silverlight, respectively); these are installed inside the web browser.

The current RIA landscape offers four platforms: HTML5/AJAX, Java, Microsoft Silverlight, and Adobe Flash Platform. Each has distinct advantages and disadvantages when applied to an RIA. As with many things in life, the key is discovering the right tool for your circumstance.

HTML 5 and Asynchronous JavaScript and XML (AJAX)

One of the easier technologies to understand (but not necessarily to implement) is AJAX. It is based on tools already familiar to web developers: HTML, DHTML, and JavaScript. The fundamental idea behind AJAX is to use JavaScript to update the page without reloading it. A JavaScript program running in the browser can insert new data into the page or change its structure by manipulating the HTML DOM without reloading a new page. Updates may involve new data loaded from the server in the background (using XML or other formats) or responses to user interaction, such as a mouse click or hover.

Early web applications used Java applets for remote communication. As browser technologies developed, other means, such as IFrames (floating frames of HTML content that can speak to each other and to servers), replaced the applets. In recent years, XMLHttpRequest was introduced into JavaScript, facilitating data transfers without the need for a new page request, applet, or IFrame.

In addition to the benefit of using familiar elements, AJAX requires no external plug-in. It works purely on the browser's capability to use JavaScript and DHTML. However, the reliance on JavaScript poses one of the liabilities of AJAX: It fails to work if the user disables JavaScript in the browser. Another issue with AJAX is that it has varying levels of support for DHTML and JavaScript in different browsers on different platforms. When the target audience can be controlled (say, for intranet applications), AJAX can be written to support a single browser on a particular platform. Many businesses today standardize their browsers and operating systems to help solve this problem. However, when the audiences are opened up (such as those for extranet and Internet applications), AJAX applications need to be tested, and often modified, to ensure that they run identically in all browsers and on all operating systems. As the number of browsers available for each operating system continues to increase, this becomes increasingly challenging. Combine that with the requirement many businesses have to support older, outdated versions of browsers, and ensuring that the code runs identically in all browsers often becomes the largest segment of application development.

Finally, HTML and JavaScript were not created with applications in mind. This means that, as a project grows in scale, the combination of code organization and maintenance can be challenging.

AJAX is not likely to go away anytime soon, and each day more high-profile AJAX applications (such as Google Maps) are launched to great acclaim. In fact, for certain classes of applications, usually smaller widgets and components, AJAX is clearly a better solution than any of its competitors due to its small size and relatively independent deployment model.

It should be noted that AJAX is not a programming model in and of itself. It is really a collection of JavaScript libraries. Some of these libraries include reusable components designed to make common tasks easier. Because AJAX lacks a centralized vendor, integrating these libraries introduces dependencies on third parties, which assumes a certain amount of risk.

The latest version of the HTML specification—HTML5—has added a number of new features, making it more competitive with other RIA technologies. It now includes support for vector-based drawing and animations, video playback, and lots of other new features that were previously possible only with a browser plug-in.

Unfortunately, HTML is still a draft proposal. The current HTML 5 specification available from www.w3.org/TR/html5 warns that the specification is still being formed, and is not currently stable. When it is eventually finalized, it will aim to reduce some of the complexity and certainly the disadvantages of working in this environment. However, at this time we must take a wait-and-see approach to identifying how organizations producing browsers adopt and implement this technology. Decision makers should remember that the previous four versions of HTML were implemented very differently by the various browser makers, and should wonder if there is reason to believe the fifth version will be any different.

Java Virtual Machine

Java—and the languages and frameworks built on Java—are the de facto standard for most enterprise server applications. However, Java has had a traditionally difficult time on the client for many reasons, including download size and perceived and real difficulties maintaining and updating the Java runtime.

Although there are certainly some success stories of Java client-side code, such as the Eclipse project (the development environment that Flash Builder itself uses), Java is still struggling to gain market share in the RIA space.

The current offering from the Java community in the RIA space is called JavaFX, a software platform designed to run across many JavaFX-enabled devices. While JavaFX has been embraced strongly by some of the Java community, it has so far received a lukewarm welcome by those not already committed to Java as their end-to-end solution. The 2.0 release for JavaFX is scheduled for the second half of 2011; perhaps with that release the promise of JavaFX will be fulfilled.

Microsoft Silverlight

As of the time of writing, Silverlight 4.0 is the currently released version of the Microsoft solution for building RIAs. It is a comprehensive offering whose many pieces include:

- **Silverlight**—A web-based application framework that provides application functionality, written in a subset of .NET.
- **XAML**—Extensible Application Markup Language. The XML-based language in which you declare user interfaces, XAML is somewhat analogous to Flex's MXML language, which you will learn about shortly.
- **Microsoft Expression Studio**—A professional design tool intended to build Windows client applications as well as web applications targeted for the Silverlight player. The suite specifically includes:
 - Expression Web—A visual HTML editor.
 - Expression Blend—A visual application that builds and edits applications for WPF (Windows Presentation Foundation) and Silverlight applications.
 - Expression Designer—A graphics editor.
 - Expression Encoder—A video encoder.

With these tools, Microsoft is promoting a workflow in which designers create compelling user interfaces with Expression (using WPF or Silverlight), and then developers implement the business and data access logic.

As of the time of writing, Silverlight plug-ins exist for most browsers on Windows and a few browsers on Mac OS X. A Linux and FreeBSD implementation written by Novell, named Moonlight, brings this content to those operating systems. The current stable release of Moonlight is 2.3, which supports Silverlight 2.0 applications. There was a recent release of a preview for Moonlight 4.0, which targets support for Silverlight 4.0 applications.

Silverlight is beginning to offer a compelling platform, especially for .NET developers who can use tools such as Microsoft Visual Studio and leverage existing knowledge to quickly create new applications. The only limitation that Microsoft seems to have at this moment is the distribution of the Silverlight runtime, which is still limited even on Windows, let alone Mac and Linux. According to riastats.com, 27 percent of all browsers do not have a Silverlight player installed, and nearly three-quarters of all machines running Linux do not have Silverlight.

Silverlight is absolutely worth watching over the coming years.

Adobe Flash Platform

One of the leading competitors in the RIA space is the Adobe Flash Platform. The platform is composed of a number of tools and technologies designed to provide rich and compelling user experiences while maximizing developer productivity and designer involvement.

While many pieces of the Flash Platform can be included as needed for different types of applications, the following sections include the ones most relevant to this book.

Adobe Flash Player and AIR

Originally written as a plug-in to run animations, Flash Player has evolved significantly over the years, with each new version adding capabilities while still maintaining a very small footprint. Over the past decade, Flash Player has gained near ubiquity, with some version of it installed on more than 99 percent of all desktops. Since 2002, Macromedia (now part of Adobe) began focusing on Flash as more than an animation tool. With the Flash 6 release, Macromedia began providing more capabilities for building applications. It found that the ubiquity of the player, combined with the power of the scripting language (ActionScript), enabled developers to build full browser-based applications and get around the limitations of HTML.

By targeting Flash Player, developers could also break away from browser and platform incompatibilities. One of the many nice features of Flash Player was that content and applications developed for any particular version of Flash Player would usually run on any platform/browser that supported that version. With very few exceptions, that remains true today.

AIR (Adobe Integrated Runtime) continues the tradition of Flash Player, but removes dependency on the browser. AIR allows content written for Flash Player, in addition to HTML and JavaScript, to run as a desktop application, effectively eliminating the need for ActionScript developers to learn additional languages should they wish to deploy their applications as more traditional applications or to non-desktop devices. Current versions of the AIR runtime are available for Windows, MacOS, Linux, and the Android Operating Systems. Support for other Mobile operating systems such as iOS, BlackBerry and Windows Phone 7 has been publicly discussed as a near future possibility.

Flash Professional

An industry-leading tool for creating engaging experiences, Flash is familiar to most designers worldwide. It provides a rich environment to create graphic assets as well as animations and motion.

Flash Professional can be used alone to create RIAs; however, the environment is intended for designers creating interactive content and may seem unfamiliar to developers who wish to build RIAs using a more traditional development methodology. Originally, this kept many

serious developers from successfully building applications on the Flash Platform, leading to the introduction of the Flex framework.

Flex

Sensing the need for RIA tools that were more developer-friendly, Macromedia created a language and a compiler that enabled developers to work with other, familiar languages from which the compiler could create applications to run in Flash Player. In 2004, Macromedia released Flex 1.0 (followed by Flex 1.5 in 2005). Adobe continued this cycle, releasing Flex 2.0, Flex 3.0, Flex 4, and Flex 4.5 in 2006, 2008, 2010, and 2011, respectively. Functionally, Flex applications are similar to AJAX applications, in that both are capable of dynamic updates to the user interface and both include the ability to send and load data in the background.

Flex goes further by providing a richer set of controls and data visualization components along with the next generation of developer tools and services to enable developers everywhere to build and deploy RIAs on the Flash Platform.

Flex builds on several key technologies and introduces several new pieces:

- **ActionScript 3.0**—A powerful object-oriented programming language that advances the capabilities of the Flash Platform. ActionScript 3.0 is designed to create a language ideally suited for building RIAs rapidly. Although earlier versions of ActionScript offered the power and flexibility required for creating engaging online experiences, ActionScript 3.0 advances the language, improving performance and ease of development to facilitate even the most complex applications with large datasets and fully object-oriented, reusable code.
- **MXML**—A declarative XML syntax that allows for the simplified creation and maintenance of user interfaces. MXML provides an easy way to visualize and implement the hierarchical nature of a complex UI without the need to directly write and maintain ActionScript code. MXML is automatically compiled to ActionScript during the process of building a Flex application.
- **Flash Player 10.x**—Building on the success of Flash Player 9, this next generation of Flash Player focuses on improving script execution and driving Flash Player to additional smaller devices. To facilitate this improvement, FP10 introduces optimizations to the ActionScript Virtual Machine (AVM), known as AVM2, a virtual machine that understands how to execute ActionScript 3.0 code regardless of the device displaying the content (for example, a mobile phone, a TV, or a desktop computer). Major improvements in runtime-error reporting, speed, memory footprints, and improved use of hardware acceleration are allowing developers for this platform to target a variety of devices well beyond the desktop/web browser. Unlike applications built with JavaScript, Flash Player is capable of using a

just-in-time (JIT) compilation process that enables the code to run faster and consume less memory, like a native application installed on your computer. More recent releases of the Flash Player have included support for high definition video, improved hardware (and GPU) acceleration, multi-monitor full screen support, and more.

- **Flex SDK**—Using the foundation provided by FP10 and ActionScript 3.0, the Flex framework adds an extensive class library to enable developers to easily use the best practices for building successful RIAs. Developers can get access to the Flex framework through Flash Builder or the free Flex SDK, which includes a command-line compiler and debugger, enabling developers to use any editor they prefer and still be able to access the compiler or debugger directly.
- **Flash Builder 4.5**—Flash Builder 4.5 is the second release of the rebranded IDE (integrated development environment) built on the success of Flex Builder 2 and 3. Providing developers with an environment specifically designed for building RIAs, Flash Builder 4.5 takes the IDE to the next level. Built on top of the industry-standard open source Eclipse project, Flash Builder 4.5 provides an excellent coding and debugging environment and promotes best practices in coding and application development. Drawing on the benefits of the Eclipse platform, it provides a rich set of extensibility capabilities, so customizations can easily be written to extend the IDE to meet specific developers' needs or preferences. Additions to Flash Builder 4.5 include custom code templates, support for development of mobile and multi-screen applications, improved designer/developer workflow, and much more.

Flash Catalyst

There is also a new version of Flash Catalyst—a tool focused on designers creating RIAs. Flash Catalyst allows designers to build applications by selecting portions of preexisting artwork created in Adobe Photoshop, Illustrator, or Fireworks and then indicating how that artwork relates to the application (scrollbar, button, and so on).

Using this tool, a designer can create pixel-perfect application shells complete with behaviors, transitions, and even data placeholders that are ready for integration and implementation by a developer using Flash Builder 4.5. The combination of these tools represents a powerful cross-platform designer-developer workflow built on the industry-standard graphics editing capabilities offered by the Adobe Creative Suite.

The latest version of Flash Catalyst allows for custom, skinnable components, the ability for developers to “protect” code, so design changes don't accidentally break functionality, and much more.

Flash Platform Server Technologies

Although beyond the scope of this book, the Flash Platform extends further to the server, providing services and functionality that RIAs can consume. Available servers and services range from workflow engines to data persistence and from messaging to video publishing and delivery.

Although you don't need any of these components to build rich experiences and applications, they do provide amazing capabilities that can extend your applications' reach and use.

What You Have Learned

In this lesson, you have:

- Explored the evolution of computer applications (pages 4–6)
- Explored alternatives to page-based architecture (pages 6–7)
- Explored the benefits of RIAs (pages 7–8)
- Compared RIA technologies (pages 8–15)

LESSON 2

What You Will Learn

In this lesson, you will:

- Create a Flash Builder project
- Understand the parts of the Flash Builder workbench: editors, views, and perspectives
- Create, save, and run application files
- Use some of the features in Flash Builder that make application development faster and easier, such as code hinting and local history
- Work in both Source view and Design view
- Use various views, such as the Package Explorer

Approximate Time

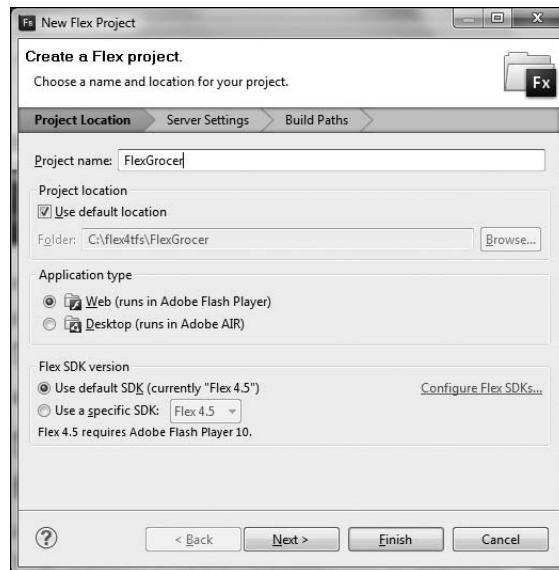
This lesson takes approximately 1 hour and 30 minutes to complete.

LESSON 2

Getting Started

You're ready to start your adventure of learning Adobe Flex, so the first thing to do is become familiar with the environment in which you will be developing your applications. This environment is Adobe Flash Builder, which is based on the Eclipse platform. The Eclipse platform is an open source integrated development environment (IDE) that can be extended. Flash Builder has extended and customized Eclipse for building Flex applications.

In this lesson, you'll become familiar with Flash Builder by building the main application files of the FlexGrocer application that you'll be working on throughout this book. While working on the FlexGrocer application, you'll learn about the Flash Builder interface and how to create, run, and save application files. You'll also discover some of the many features Flash Builder offers to make application development easier.



Creating a new project in Flash Builder

Getting Started with Flex Application Development

Before you can build a building, you must lay the foundation. This lesson is the foundation for further Flex development. You will finish this lesson knowing how to manipulate Flash Builder in ways that make the process of Flex development easier and faster. Along the way, you will create the main application file that defines the FlexGrocer application.

Part of the study of any new body of knowledge is learning a basic vocabulary, and in this lesson you will learn the basic vocabulary of both Flex development and Flash Builder. You'll understand terms such as *view*, *perspective*, and *editor* in relationship to the Flash Builder workbench. Also, you'll understand the terms describing the processes that transform the text you enter in Flash Builder into the type of file you can view with your browser using Flash Player.

Creating a Project and an MXML Application

In this first exercise, you will create a Flex application. To do so, you must first create a project in Flash Builder. A *project* is nothing more than a collection of files and folders that help you organize your work. All the files you create for the FlexGrocer application will be in this project. You'll also see that you have two choices when working with an application file: You can work in either Source view or Design view. In most cases, the view you choose will be a personal preference, but at times some functionality will be available in only one view.

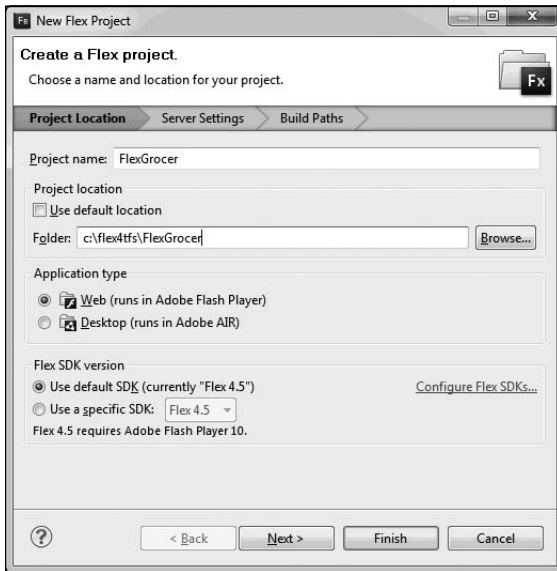
Also in this exercise, you will run the Flex application. You'll discover how the code you write is turned into a SWF file that is viewed in a browser.

- 1** Start Flash Builder: On Windows choose Start > Programs > Adobe > Adobe Flash Builder 4.5; on Mac OS open the Flash Builder application from the Adobe Flash Builder 4.5 folder in your Applications directory.

This is most likely the way you will start Flash Builder. You may have already installed Eclipse on your computer and added the Flex functionality using the plug-in configuration. In that case, you need to open Eclipse as you have before, and switch to the Flash perspective.

- 2** Choose File > New > Flex Project. For the Project name, enter **FlexGrocer**. Deselect the "Use default location" check box, and for the Folder location enter *driveroot:/flex4tfs/FlexGrocer*.

- * NOTE:** *Driveroot* is a placeholder for the name of the root drive of the operating system you are using, either Windows or Mac. Replace *driveroot* with the appropriate path. Also, note that the directory name is case sensitive.



The project name should reflect the files contained in the project. As you continue your work with Flash Builder, you'll soon have many projects, and the project names will help remind you which files are in each project.

Do not accept the default location entry. The default uses your Documents directory and places files very deep in the directory structure. For simplicity's sake in this project, you are putting your work files right on the root drive.

Flash Builder lets you choose whether to use the most recent compiler (the default choice) or one from a previous version, by selecting the appropriate "Flex SDK version" radio button. For this application, you should use the Flex 4.5 SDK and compiler.

- 3** Click Next.

- 4 Leave the Server technology set to None/Other and the output folder for the compiled Flex application as bin-debug. At this time, there is no need to change this default. Click Next.

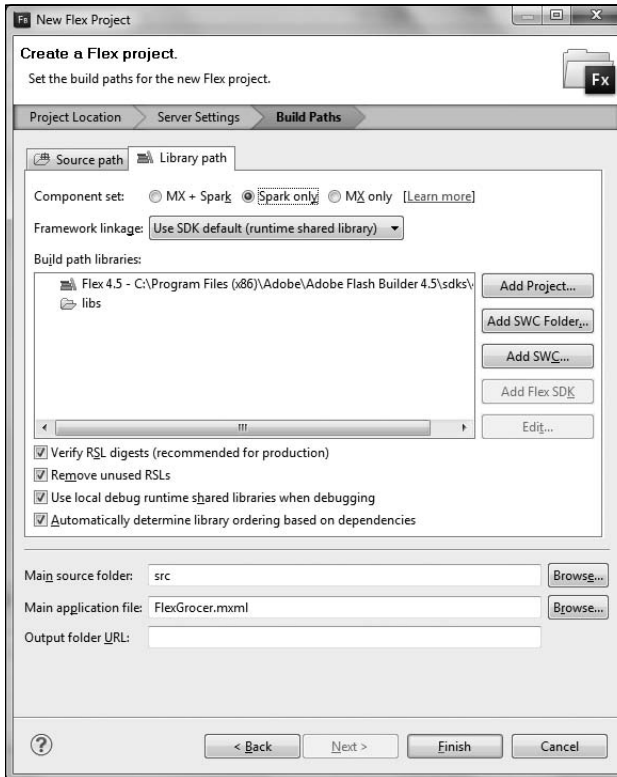


- 5 Across the top of the dialog box, there are three radio buttons next to a label named Component set. Choose the “Spark only” option.

Flash Builder 4.5 allows you to work with either an older style set of components named mx, the newer style components named spark, or a hybrid of the two. In this book, you will work with the latest components from the spark set only. The mx components are described thoroughly in a variety of materials including the “Flex 3 Training from the Source” series.

6 Ensure that **FlexGrocer.mxml** is set as the main application filename.

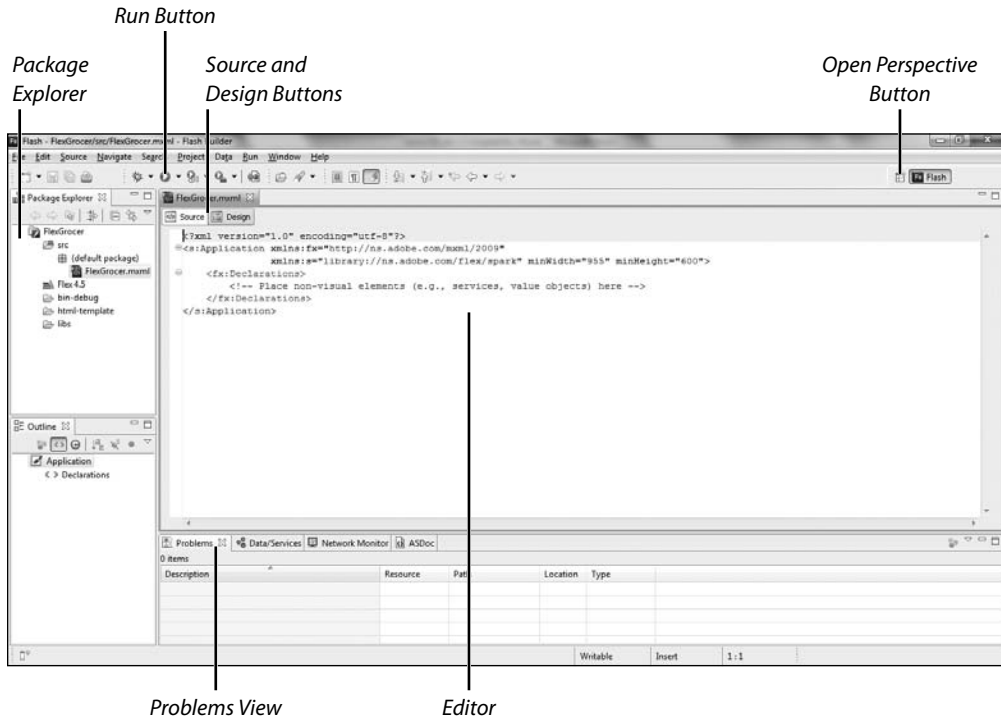
By default, Flash Builder gives the main application file the same name as your project name. Flash Builder automatically creates the main application file for you and includes the basic structure of a Flex application file.



*** NOTE:** MXML is a case-sensitive language. Be sure to follow the case of the filenames in tags shown in this book.

7 Click Finish and see the project and application file you created.

Here you see your first Flex application. Currently the application is displayed in Source view. In later lessons, you will also look at this application in Design view.



The default application file contains some basic elements. The first line of code

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

is an XML document-type declaration. Because MXML is an XML standard language, the document declaration should be included in the code.

Starting with the second line of code,

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="955" minHeight="600">
```

you will see definitions for a Flex main application and their associated namespaces. A minimum height and width for the application are also defined. The `<s:Application>` tag represents the outside container, the holder of all the content in the Flex application. You can have only one `<s:Application>` tag per Flex application.

Inside the `<s:Application>` tag are two attribute/value pairs that refer to URLs, such as `xmlns:fx="http://ns.adobe.com/mxml/2009"`. These declarations are XML namespaces: They say where the definitions for the classes you will use in your application are and how you refer to those classes. In Flex 4.5 applications, it is common to have the two namespaces provided, as you are likely to reference three particular libraries of classes: Flex language tags, represented by the `fx` namespace and Flex Spark components, represented by the `s` namespace.

The Flex language tags represent elements that are needed for Flex 4.5 applications, but these elements are not actually classes within the Flex 4.5 SDK. You will find the `<fx:Declarations>` tag one of the more frequently encountered language tags. Declarations will be explained in more detail when events are discussed. The Spark classes represent the set of components, Buttons, Checkboxes, and so on used in Flex 4.5.

*** NOTE:** In XML nomenclature, the part of the attribute between the colon (:) and the equal sign (=) is known as the prefix, and the quoted string after the equal sign is known as the Universal Resource Identifier (URI). So, given `xmlns:s="library://ns.adobe.com/flex/spark"`, `s` is the prefix, and `library://ns.adobe.com/flex/spark` is a URI.

In a configuration file called `flex-config.xml`, an association is made between these URIs and an associated manifest file. Each manifest file contains a delineation of all the legal tags that can be used with that particular prefix and of the location of the class to which each tag will refer. In a standard installation of Flash Builder on a PC, the manifest files are located in *installationdirectory/Adobe/Adobe Flash Builder 4.5/sdks/4.5.0/frameworks*. On a Mac, the manifest files are found at *installationdirectory/Adobe Flash Builder 4.5/sdks/4.5.0/frameworks*. The `fx` namespace is pointing to the `mxml-2009-manifest.xml`, the `s` namespace points to the `spark-manifest.xml`, and the `mx` namespace points at the `mx-manifest.xml`.

*** NOTE:** If you look in the manifest file for the `fx` namespace (`mxml-2009-manifest.xml`), you'll notice the conspicuous absence of Declarations as one of the listed classes. In fact `fx:Declarations` is not a reference to a class, so much as it is a compiler directive, instructing the compiler how to associate metadata with the ActionScript class created from the MXML. It's more important to note that the other two manifest files do indeed contain references to all the classes you'll make use of when using their namespaces.

Part of the spark-manifest.xml file is shown here.

```

- <componentPackage>
  <!-- Flex4 Framework -->
  <component id="AddAction" class="spark.effects.AddAction" />
  <component id="Animate" class="spark.effects.Animate" />
  <component id="AnimateColor" class="spark.effects.AnimateColor" />
  <component id="AnimateFilter" class="spark.effects.AnimateFilter" />
  <component id="AnimateTransitionShader" class="spark.effects.AnimateTransitionShader" />
  <component id="AnimateTransform" class="spark.effects.AnimateTransform" />
  <component id="AnimateTransform3D" class="spark.effects.AnimateTransform3D" />
  <component id="Animation" class="spark.effects.animation.Animation" />
  <component id="Application" class="spark.components.Application" />
  <component id="ArrayCollection" class="mx.collections.ArrayCollection" lookupOnly="true" />
  <component id="ArrayList" class="mx.collections.ArrayList" lookupOnly="true" />
  <component id="AsyncListView" class="mx.collections.AsyncListView" lookupOnly="true" />
  <component id="BasicLayout" class="spark.layouts.BasicLayout" />
  <component id="BevelFilter" class="spark.filters.BevelFilter" />
  <component id="BitmapImage" class="spark.primitives.BitmapImage" />
  <component id="Block" class="spark.layouts.supportClasses.Block" />
  <component id="BlurFilter" class="spark.filters.BlurFilter" />
  <component id="BorderContainer" class="spark.components.BorderContainer" />
  <component id="Bounce" class="spark.effects.easing.Bounce" />
  <component id="Button" class="spark.components.Button" />
  <component id="ButtonBar" class="spark.components.ButtonBar" />
  <component id="ButtonBarButton" class="spark.components.ButtonBarButton" />
  <component id="ButtonBarHorizontalLayout" class="spark.components.supportClasses.ButtonBarHorizontalLayout" />
  <component id="ButtonBase" class="spark.components.supportClasses.ButtonBase" />
  <component id="CallAction" class="spark.effects.CallAction" />

```

Finally, a minimum height and width are defined for the application. By specifying these, Flash Player will know whether the browser that the application is running in is large enough to fit the application. If the browser is not large enough, scroll bars need to be added to allow the user to access the rest of the application.

Understanding the Flash Builder Workbench

Before you do any more work on your application file, you need to become more familiar with the Flash Builder *workbench*, which is everything you see in Flash Builder. In this exercise you'll learn what views, editors, and perspectives mean in the workbench.

- 1 Close the current editor by clicking the X on the right side of the FlexGrocer.mxml editor tab. All editors have a tab at the top left of the editor area.

Whenever you open a file, it is opened in the workbench in what is called an *editor*. You just closed the editor containing the FlexGrocer.mxml file. You can have many editors open at once in the workbench, and each will contain a file with code in it.

- 2 Open the editor containing the FlexGrocer.mxml file from the Package Explorer by double-clicking the filename.

You can also open the file by right-clicking the filename and choosing Open.

- 3 Make the editor expand in width and height by double-clicking the editor tab.

Sometimes you'll want to see as much of your code as possible, especially because Flash Builder does not wrap text. Simply double-clicking, the editor tab expands (maximizes) the editor in both width and height, showing as much code as possible.

- 4 Restore the editor to its previous size by double-clicking the tab again.

As you see, you can easily switch between expanded and non-expanded editors.

- 5 Click the Design view button in the editor to view the application in Design view. Your application now looks more like it will look to an end user.

The workbench looks radically different in Design view, which allows you to drag user interface controls into the application. You will also be able to set property values in Design view.

- 6 Return to Source view by clicking the Source view button in the editor.

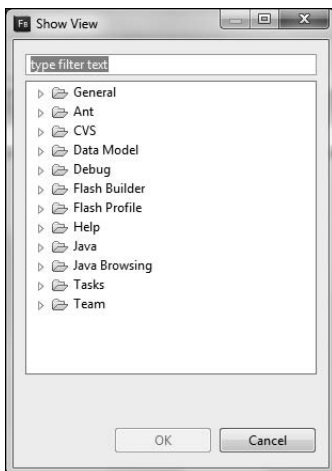
Most frequently, you will be using Source view in this book, but some tasks are better performed in Design view.

- 7 Close the Package Explorer by clicking the X on the Package Explorer tab. Just like editors, views have tabs at their top left.

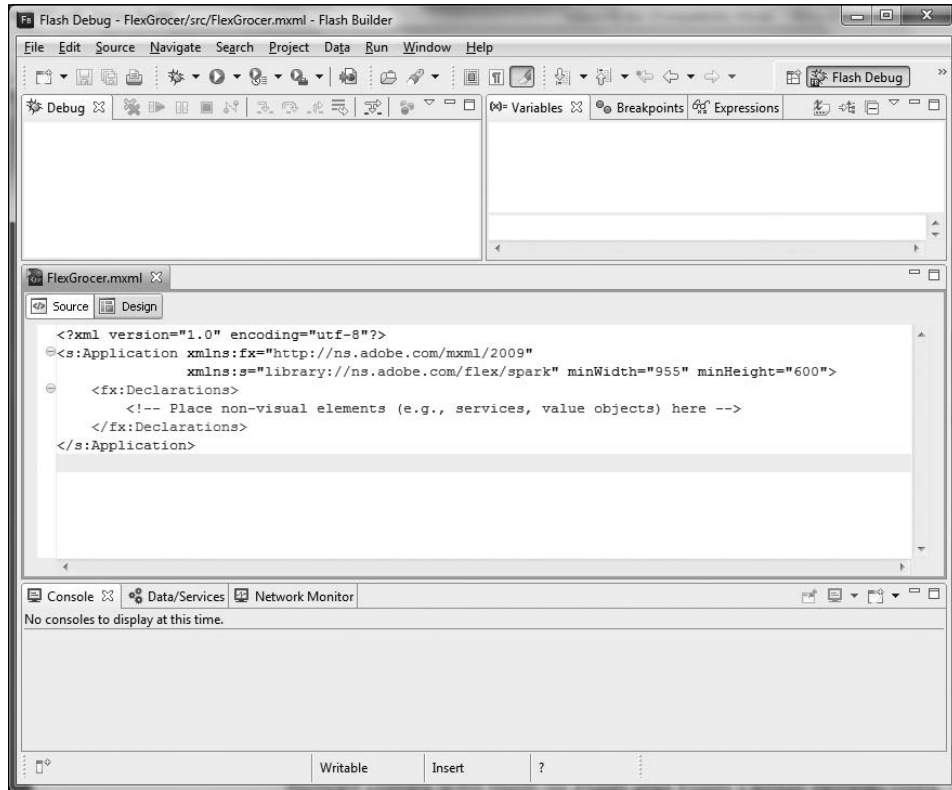
In Flash Builder, the sections displaying content are called *views*.

- 8 Reopen the Package Explorer by choosing Window > Show View > Package Explorer.

After you close a view, you can reopen it from the Window menu. There are many views; in fact, if you choose Window > Show View > Other, you'll see a window listing all the views.



- 9 Click the Open Perspective button just above the top right of the editor, and choose the Flash Debug perspective.

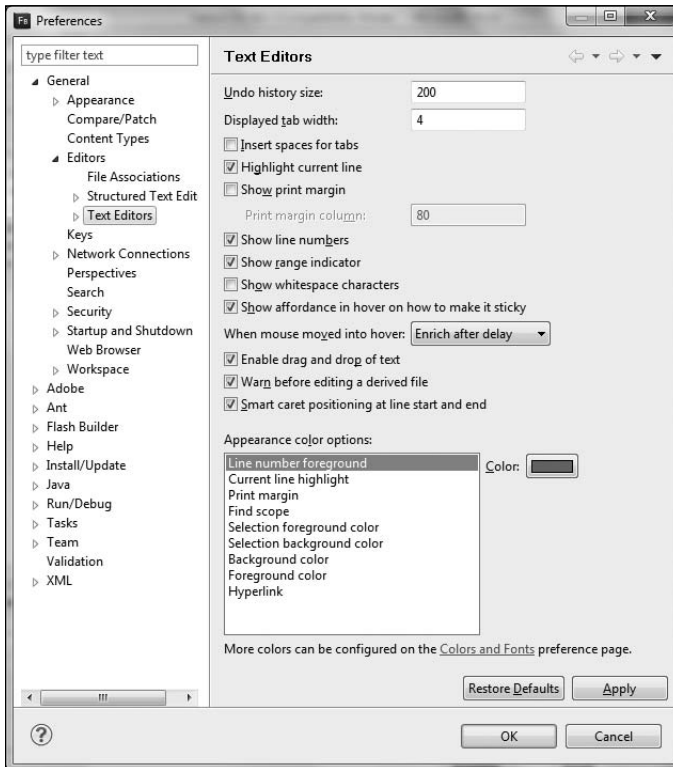


A *perspective* is nothing more than a layout of views that you want to use repeatedly. Flash Builder comes with built-in Flash and Flash Debug perspectives. You can adjust the layout of views in Flash Builder in any way that works best for you, and save it as a perspective by choosing Window > Save Perspective As. Once it's saved, you can switch to that perspective from the Open Perspective menu at any time.

- 10 Return to the Flash perspective by clicking the Flash perspective button.

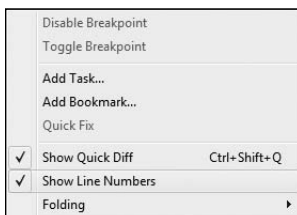
As you can see, it's easy to switch between perspectives. Later in the lesson, you'll use the Debug perspective and discover its many helpful options.

- 11** If they are not showing, turn on code line numbers by choosing Window > Preferences (Flash Builder > Preferences on Mac). In the dialog box, click the disclosure triangles to the right of General and then click Editors to expand the menus. Finally, click Text Editors and select the check box for “Show line numbers.”



Line numbers are useful because Flash Builder reports errors using line numbers.

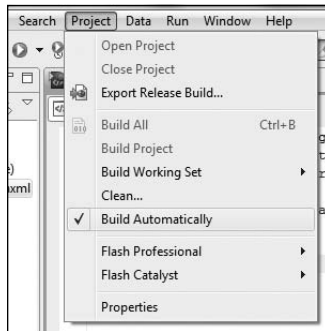
- TIP:** You can also turn on line numbers by right-clicking in the marker bar of the editor and choosing Show Line Numbers. The marker bar is the area just to the left of where the code is displayed in the editor.



Running Your Application

In the first exercise, you created your project and an application page. Before you had a chance to run the application, you took a tour of the Flash Builder workbench. You will now get back to your application. You will run it, add code to it, and learn the basics of file manipulation.

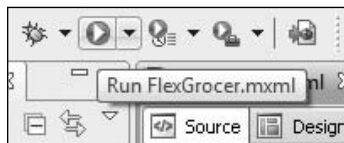
- 1 Open the Project menu. Be sure the Build Automatically option has a checkmark in front of it.



When Build Automatically is selected, Flex continuously checks for saved files, compiles them upon saving, and prepares them to run. Even before you run your application, syntax errors are flagged, which does not occur if Build Automatically is not selected.

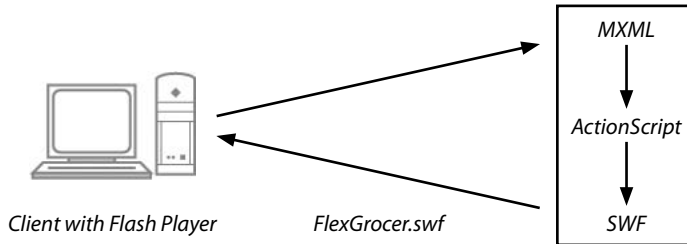
- ▶ **TIP:** As your applications grow more complex, you might find that having the Build Automatically setting selected takes too much time, in which case you should deselect the setting. The build will happen only when you run your application or you specifically choose Build Project from the menu.

- 2 Run your application by clicking the Run button. You will not see anything in the browser window when it opens.



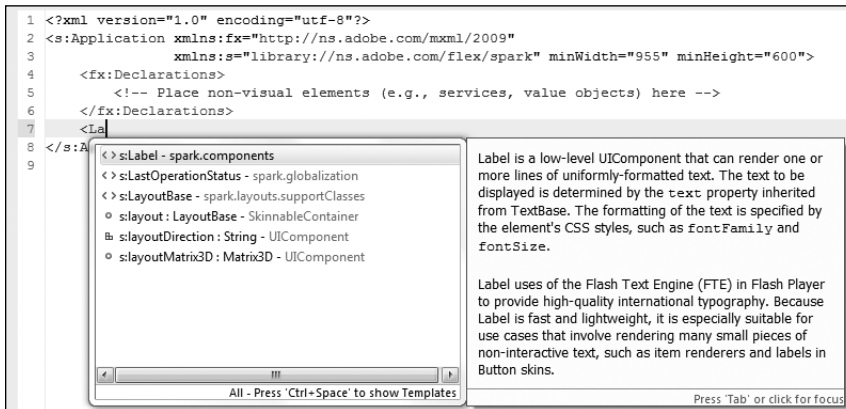
You have now run your first Flex application, and it wasn't that interesting. In this case, the skeleton application contained no tags to display anything in the browser. But you did see the application run, and you saw the default browser open and display the results, uninteresting as it was.

- * NOTE:** What exactly happened when you clicked the Run button? A number of processes occurred. First, the MXML tags in the application file were translated to ActionScript. ActionScript was then used to generate a SWF file, which is the format Flash Player understands. The SWF file was then sent to Flash Player in the browser.



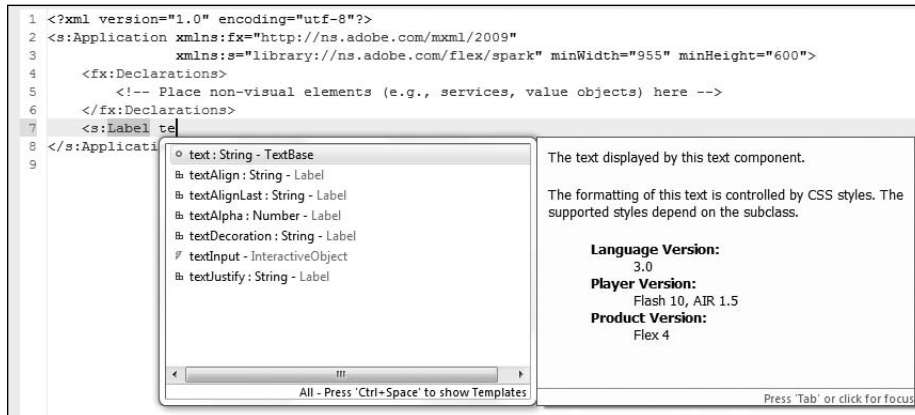
- 3 Close the browser and return to Flash Builder.
- 4 Add an `<s:Label>` tag by placing the cursor after the closing `</fx:Declarations>` tag; press Enter/Return; enter the less-than sign (`<`). You will see a long list of tags. Press the letter `L` (upper- or lowercase) and select the Label tag by highlighting it and pressing Enter or by double-clicking it.

This is an example of code hinting, which is a very helpful feature of Flash Builder of which you should take advantage.



- 5 Press the spacebar, and you'll see a list of options, including properties and methods, which you can use with the `<s:Label>` tag. Press the letter `t` and then the letter `e`; then select the `text` property.

Code hinting shows only the elements that relate to the selected tag. So, seeing the text element appear in this list indicates that it is a valid attribute for this tag. Not only can you select tags via code hinting, but you can also choose attributes that belong to those tags.



*** NOTE:** In these two instances of code hinting, both the desired options happen to be at the top of the list. If the options were not at the top, you would select the desired option either by pressing the Down Arrow key and then pressing Enter or by double-clicking the selection.

6 Enter **My First Flex Application** for the value of the text property. Be sure that the text is in the quotes supplied by code hinting.

Given that MXML is an XML language, it is required to follow all XML rules and standards. Proper XML formatting dictates that the value of any attribute be placed in quotes.

7 End the tag with a slash (/) and a greater-than sign (>).

Check to be sure that your code appears as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="955" minHeight="600">
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
  <s:Label text="My First Flex Application"/>
</s:Application>

```

- * NOTE:** The code in this example places the `minWidth` and `minHeight` attribute/value pairs of the `Application` tag on a separate indented line. The entire `Application` tag could have been on one line; whether or not to add line breaks to code is a matter of personal preference. Some developers like the look of placing each attribute/value pair on a separate indented line.

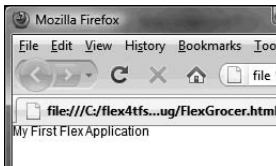
Proper XML syntax gives you two ways to terminate a tag. You just used one of them—to place a slash at the end of the tag, which is called a self-closing tag. The other option is to use the slash in front of the tag name, which is completely typed out again, as follows:

```
<s:Label text="My First Flex Application">
</s:Label>
```

You usually use the self-closing tag, unless there is a reason to place something inside a tag block. For example, if you want to place the `<s:Label/>` tag inside the `<s:Application>` tag block, you have to terminate the `</s:Application>` tag on a separate line.

- 8** Save the file and run it. The text “My First Flex Application” appears in your browser.

Finally, you get to see something displayed in your new Flex application.



The `<s:Application>` tag comes with a default background color of white (`#FFFFFF`). You will learn more about adjusting styles in Lesson 16, “Customizing a Flex Application with Styles.”

- 9** Change the value of the text property from “My First Flex Application” to **New Text**. Save the file and run it.

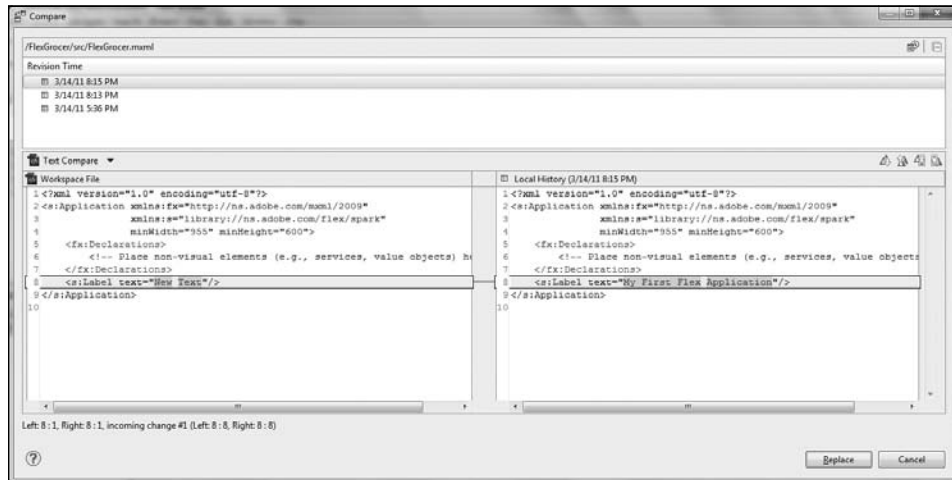
The next step shows another helpful feature of Flash Builder, but to see this feature you must have at least two saved versions of the file, which is why you changed the text and saved another version of the file.

- 10** Right-click in the editor, and from the contextual menu choose `Replace With > Local History`.

A large dialog box appears.

- 11** Compare the current version of your file, which is located on the left side of the dialog box, to the older version on the right. Select the first item in the list. A history of the last 50 versions of your file is kept at the top of the dialog box. Click `Replace` to bring back your original text, which reads “My First Flex Application”.

You will find this feature very helpful when you want to roll back to a previous version of code.

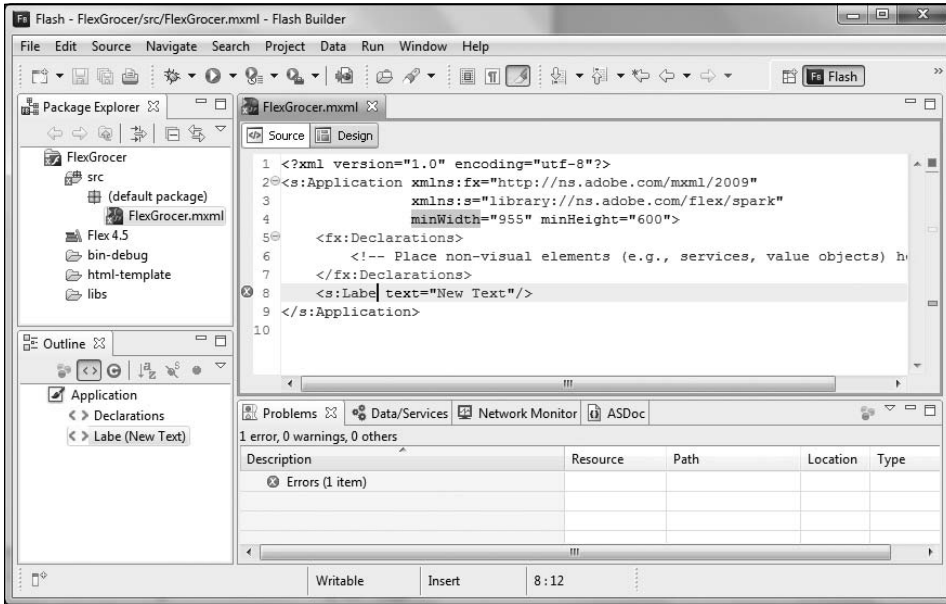


▶ **TIP:** You can alter the settings for Local History by choosing Window > Preferences. Then from the dialog box, choose General > Workspace and click Local History.

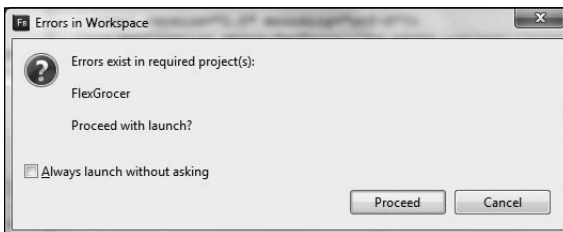
12 Purposefully introduce an error into the page by removing the ending *l* from *Label*, changing the `<s:Label>` tag to `<s:Labe>`, and save the file. This will cause an error, because the compiler can find the Label class in the `s` namespace, but not a Labe class.

After you save the file, the compiler checks your code. The error is found and reported in two ways. First, a small white X in a red circle will appear next to the line of code in which the coding mistake is located. Also, a description of the error appears in the Problems view.

▶ **TIP:** You can place the pointer over the Red circle by the line number to see the complete description. You can also double-click the error listed in the Problems view, and the pointer will then appear at the appropriate line of code in the editor.



- 13 Run the application. You'll see the following warning, telling you there are errors in your project and prompting you to confirm that the launch should continue. In this case, click Cancel.



Because of the compile-time error, Flash Builder will not be able to compile the application with the latest change. If you click Proceed in this dialog box, Flash Builder will run the last successfully compiled version of your application.

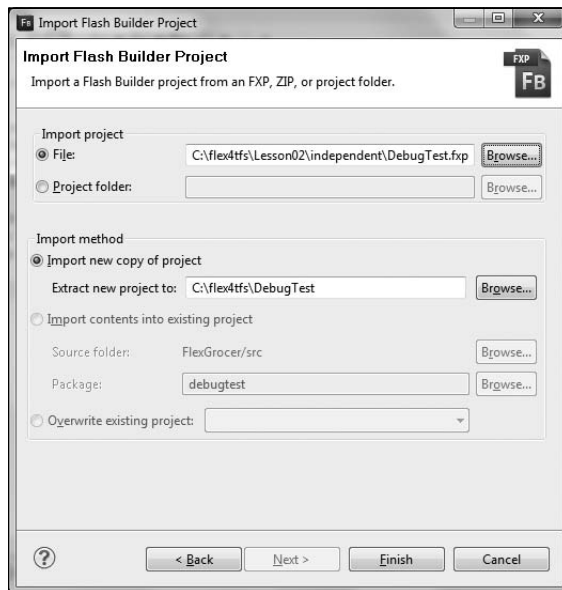
- 14 Correct the error, save the file, and run it to be sure that everything is again working properly. Close the file.

Exploring the Flash Builder Debugger

As you build applications, things will sometimes not work properly or will perhaps throw errors. To help you understand what is causing these problems, Flash Builder has an interactive debugger. The debugger lets you set breakpoints in the code and inspect various property and variable values at the point where the execution of the code stops. You can also use the debugger to step through the code, so you can see how those values change over time.

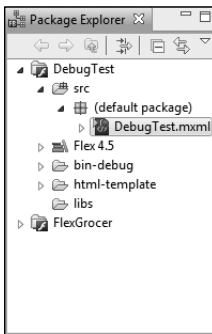
▼ IMPORTANT: If you haven't done so already, please follow the instruction in the appendix on *Installing Lesson Files*.

- 1 From the main menu of Flash Builder, choose File > Import > Flash Builder > Flash Builder Project. This project file contains a small application with a button, a label, and ActionScript code that will add two numbers and display the results. Flash Builder has the ability to import pre-existing projects packaged in the FXP format as a stand-alone file.
- 2 In the Import Flash Builder Project dialog box that opens, click the first Browse button on the right of the screen. Navigate to the flex4tfs/Lesson02/independent directory, and select the DebugTest.fxp file. Set the location for “Extract new project(s) to” as **driveroot:\flex4tfs\DebugTest**.

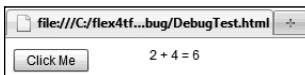


If the DebugTest directory does not exist (and it probably doesn't, unless you have done these steps before), it will be created for you. If it does exist, and you want to replace the previous version, be sure to select the “Overwrite existing project” radio button in the “Import method” section of the dialog box. If you are overwriting an existing version, you will be prompted to confirm that the previous version is to be replaced. At this prompt, you will need to click OK to continue.

- 3 Click Finish.
- 4 In Flash Builder, notice that a new DebugTest project has been created. Expand the src and default package nodes to find the DebugTest.mxml file. Open DebugTest.mxml by double-clicking it.

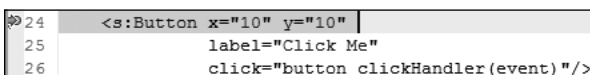


- 5 Run the application and click the Click Me button. You should see a line of text appear next to the button, reading “2 + 4 = 6”.



When you clicked the button, the event handler defined on line 26 was executed, calling the `button_clickHandler()` method. This method defines two integer variables (`numTwo` and `numFour`) and passes them to a function that adds the integers together and then displays the results in a label. The ActionScript syntax, event handlers, and datatypeing variables will all be covered in detail in the next few chapters.

- 6 In Source view, double-click the line number 24 to set a breakpoint on that line. You need to double-click the number itself, not merely that line of code. When the breakpoint is set, you should see a blue circle appear to the left of that line number.



Flash Builder may move your break point to line 26. Flash Builder will always look for the next executable line of code. In this case, that is line 26 where the click occurs.

When the application runs in Debug view, and a user clicks the button, the application will stop executing at the line with the breakpoint. You will be able to debug from that point in the application. You can set breakpoints at any line of executable code in an application, such as an event handler in MXML, or a line of ActionScript in the Script block.

- 7 Launch the application in Debug view, by clicking the button that looks like a bug (located to the right of the Run application button).

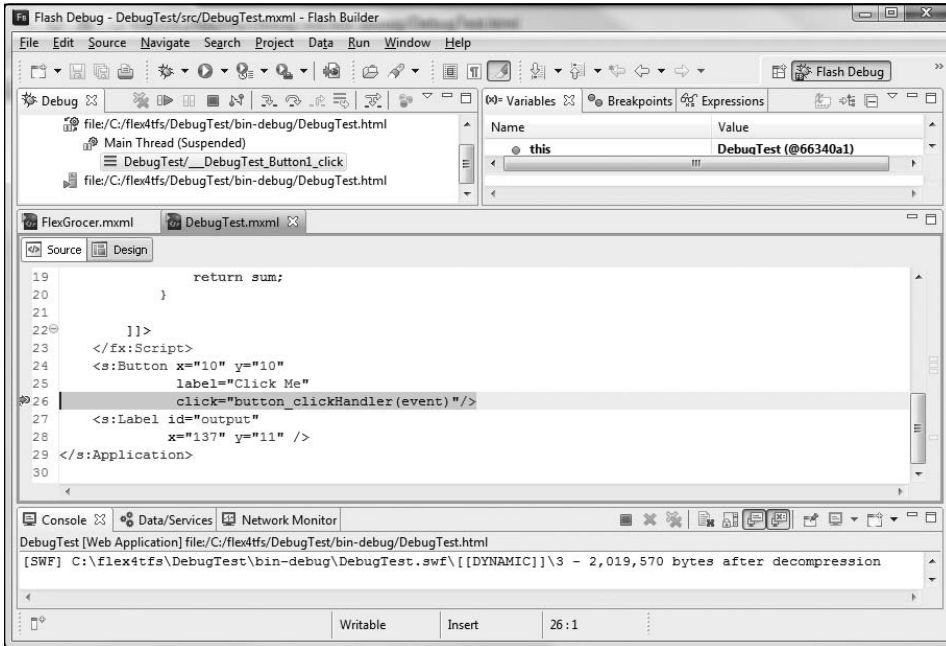


- 8 When the browser launches the application, click the button labeled Click Me. This time the application runs until it reaches the breakpoint, then control is passed from the browser to Flash Builder. If Flash Builder is in the Flash perspective, you will be asked if you want to switch to the Debug perspective. Select the Remember My Decision check box, then click Yes.



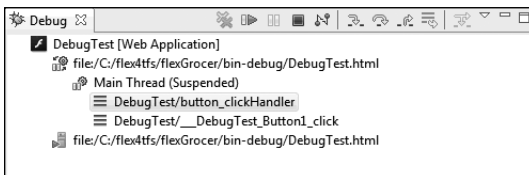
As you learned earlier in this chapter, Eclipse (and therefore Flash Builder) uses perspectives to group together sets of views. By default, the Debug perspective adds four views above the code. To the left is the Debug view, which shows where the application encountered the breakpoint. This view also has buttons to continue running the application, to stop debugging, and to step into the code, step over a line in the code, or return to the place that called the current function.

To the right of the Debug perspective are three tabbed views. The Variables view shows the current state of the variables, the Breakpoints view shows all the breakpoints, and the Expressions view shows any watch expressions you have added. You will explore these views and buttons in the next few steps.



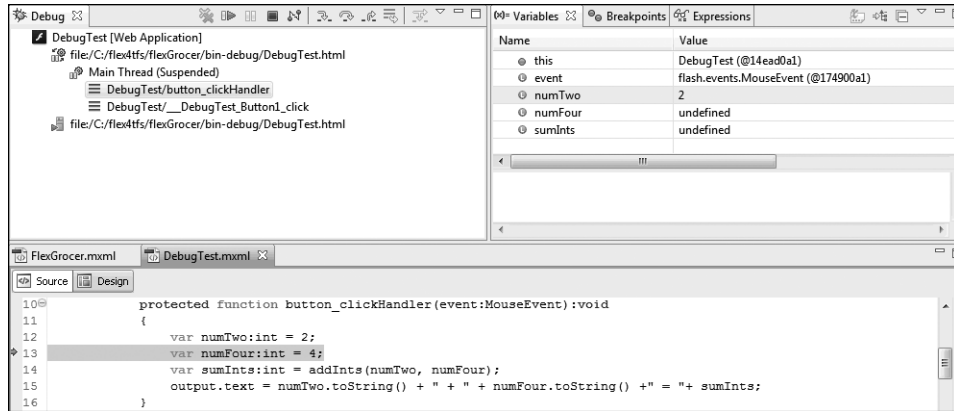
- In the Debug view, click the Step Into button, which will move control to the `button_clickHandler()` method.

The Debug view is showing you that you're looking at the `button_clickHandler()` method. The Variables view will still show this event, as it did initially, but three new items are visible there as well, representing the three variables defined locally to the function. Since you have not yet executed the line of code that instantiates and sets the values of those variables, they currently have a value of `undefined`.



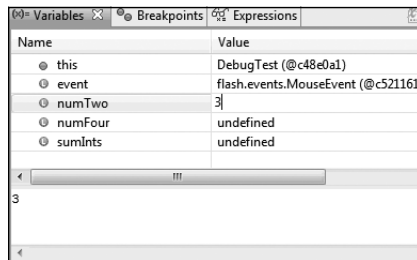
- Click the Step Over button and notice that control is moved to the next line of executable ActionScript. Control stops immediately before this line executes. Click Step Over again to execute the code on the line that instantiates a variable named `numTwo` and assigns it a value of 2.

Now that the `numTwo` variable has a value, you can see the value in the Variables view.



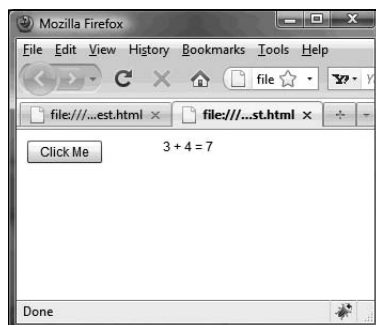
- 11** In the Variables view, click the value numTwo, and change it from 2 to 3.

The Variables view lets you change the values of variables and see what effect the change has on the application. Changing this value won't change the underlying code that set the value, but it will let you see what happens if the value is different.



- 12** In the Debug view, click the Resume button to allow the application to continue executing.

This time, the label reads “3 + 4 = 7”, as we changed the value of the numTwo variable from 2 to 3.



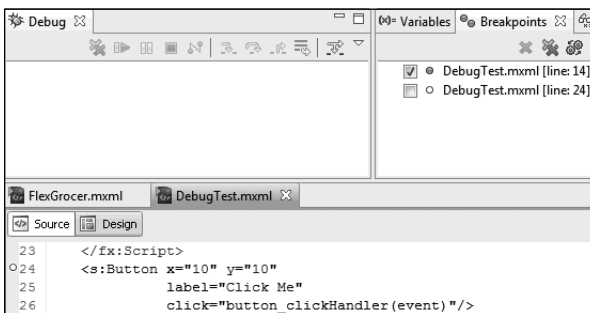
- 13** In the Debug view, click the Terminate button (red square) to stop this debugging session. Double-click the line number 14 to set a breakpoint there.

You now have breakpoints set at lines 14 and 24.

```
11      {
12          var numTwo:int = 2;
13          var numFour:int = 4;
14          var sumInts:int = addInts(numTwo, numFour);
15          output.text = numTwo.toString() + " + " + numFour.toString() + " = "+ sumInts;
16      }
17      private function addInts( value1:int, value2:int ):int{
18          var sum:int = value1 + value2;
19          return sum;
20      }
21
22  }]}>
23  </fx:Script>
24  <s:Button x="10" y="10"
25          label="Click Me"
26          click="button_clickHandler(event)"/>
```

- 14** Click the Breakpoints tab to see that view and notice that you now have breakpoints set on lines 14 and 24. You can turn on or off breakpoints by clicking the check boxes next to each breakpoint. At this point, we no longer want to use the breakpoint at line 24, so deselect its check box.

Deselecting the check box leaves the breakpoint in place but instructs Flash Builder to ignore it for now. You'll notice that the icon on that line of the code has changed from a blue circle to a white circle. If you want to completely remove a breakpoint, you can either double-click its line number again, or right-click the breakpoint in the Breakpoints view and choose Remove.



- 15** Run the application in Debug view again, and click the Click Me button when the application starts.

Notice that this time, the execution stops at the breakpoint on line 14, and that the `numTwo` and `numFour` variables are already populated.

- 16** Click the Step Into button to step into the `addInts()` function.

Notice that the Debug view shows a click on the button called the `button_clickHandler()`, which in turn has now called `addInts()`. Also notice that the Variables view is showing another set of variables instead of `numTwo` and `numFour`, which were variables local to the `button_clickHandler()` method. It is now showing `value1` and `value2`, the arguments to the method.

- 17** Click the Step Over button two times.

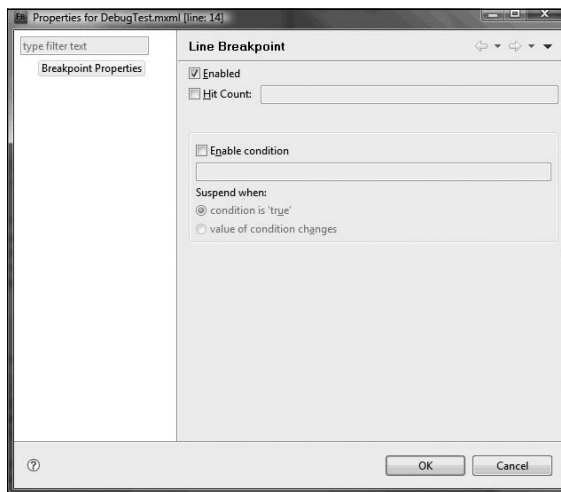
As it did the previous times you used Step Over, the debugger executes the next line, this time populating the `sumInts` variable with the value 6.

- 18** Click the Step Return button. Then click the Step Over button.

Notice that control returns to the `button_clickHandler()` method, and that the `sumInts` variable is now properly populated with the value 6, as it was computed in the `addInts()` method.

Congratulations! You know how to use the debugger. This will be extremely useful as you work through the exercises in this book and as you develop real-world Flex applications.

There's one more interesting feature available with breakpoints in Flash Builder: conditional breakpoints. You can enable conditional breakpoints by right-clicking a breakpoint (either next to the line numbers, or in the Breakpoints view), and choosing Breakpoint Properties. From the Breakpoint Properties view, you can enable or disable breakpoints. You can also set conditions. For example, you can set a breakpoint to fire only if a variable has a certain value, or when a potential breakpoint is encountered a certain number of times. While this feature may not be terribly useful as you first start your explorations in Flex, you'll find it invaluable as you build more complex applications.



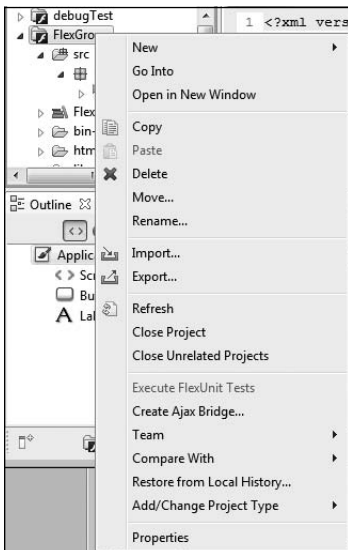
*** NOTE:** Teaching object-oriented programming is not the focus of this book, but to be an effective Flex developer you must have at least a basic understanding of object-oriented terminology and concepts. For instance, the tags you’ve seen—such as `<s:Application>`, `<s:Label>`, and `<s:Text>`—actually refer to classes. The *Adobe Flex 4.5 MXML and ActionScript Language Reference* (sometimes referred to as ASDoc) is the document that lists these classes, their properties, their methods, and much more.

Getting Ready for the Next Lessons

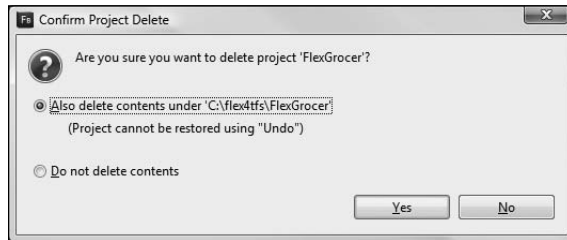
As you go forward through the rest of the book, you’ll need certain files, such as graphics, to complete the rest of the application. In the same way you imported a project from an FXP file to learn about debugging, you’ll also import an FXP file that will be the basis for the application you work on throughout this book.

When you import a project into Flash Builder, the IDE makes a determination if you already have a project with the same unique identifier (UUID). If you do, it will allow you to overwrite the existing project with the newly imported one. If not, and you already have a project of the same name as the one you are importing, it will ask you to rename the newly imported project. To avoid any confusion, you’re going to completely delete the project you had created previously in this lesson, and then import a nearly identical project, which includes some graphics that will be used throughout the rest of the book.

- 1 In the Package Explorer, right-click the FlexGrocer project, and choose Delete.

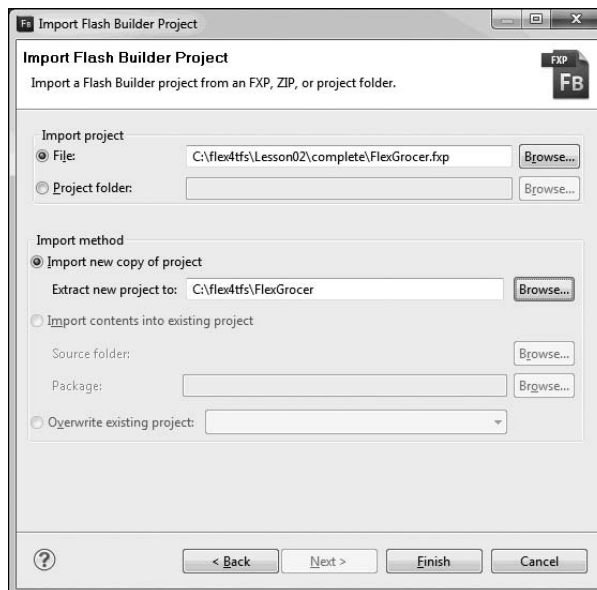


- 2 A dialog box will appear asking if you want to delete only the Flash Builder project, or also the files for the project from the file system. Select the radio button to also delete contents.



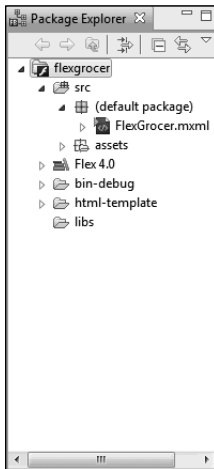
The project and related files will be removed from the file system.

- 3 From the main menu of Flash Builder, choose File > Import > Flash Builder Project.
- 4 In the Import Flash Builder Project dialog box, click the first Browse button on the right of the screen. Navigate to flex4tfs/Lesson02/complete and select the FlexGrocer.fxp file. Set your project directory as the location for "Extract new project to".



5 Click Finish.

You are now ready to continue through the rest of the book. If you care to verify that the project imported properly, look in the Project Explorer and confirm that there is now an assets directory in your project.



What You Have Learned

In this lesson, you have:

- Created a project to organize your application files (pages 18–24)
- Toured the pieces of the Flash Builder workbench (views, editors, and perspectives) used to create application files (pages 24–28)
- Run and modified application files while using code hinting and local history to produce the code for those files (pages 28–33)
- Learned about debugging an application with the Flash Builder debugger (pages 34–41)
- Prepared for the next lessons (pages 41–43)

LESSON 3

What You Will Learn

In this lesson, you will:

- Use containers
- Lay out an application in Source view
- Work with constraint-based layouts
- Work with view states
- Control view states
- Lay out an application in Design view
- Refactor code as needed

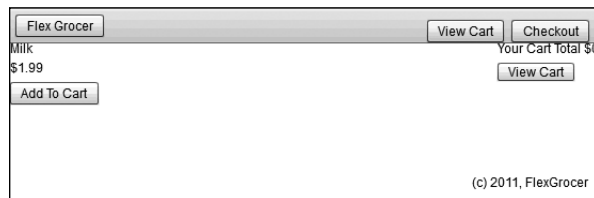
Approximate Time

This lesson takes approximately 1 hour and 30 minutes to complete.

LESSON 3

Laying Out the Interface

Every application needs a user interface, and one of the strengths of Adobe Flash Builder is that it enables developers to lay out their application's interface with ease. In this lesson, you will learn about containers and layout objects in Flex, what differentiates them, and how to use them when laying out your applications. Finally, you will use view states to make the applications dynamically change to react to users' actions.



The user interface for the e-commerce application

Learning About Layouts

Almost all positioning of components in Flex is done using containers and layout objects.

Working with a kitchen analogy for the moment, you can think of the container as a food processor without a blade. There are different food processors with different features on the market, and you need to choose one that works best for your application.

You can think of layout objects as blades that can be inserted into a food processor to slice, dice, and chop. Neither the processor nor the blade is useful without the other, but when they're assembled, they're a powerful tool. The same is true of containers and layout objects.

Understanding Containers

On a technical level, containers are simply a special type of component that contains and groups other items. These items are collectively referred to as *children* or, more specifically, as layout elements (which is just a broad term for components such as Buttons, Checkboxes, and the like) and graphical elements (such as squares, circles, and so on). Although containers know how to group and keep these items together, they do not know the position or order of those items on the screen. When you select a container type to use, you will do so based on a number of criteria; however, the most fundamental is its ability to be skinned.

Skining is the process of defining the visual appearance of a component. In terms of a container, you can think of the visual appearance as including things such as backgrounds, borders, drop shadows, and so on. Some containers in Flex can be skinned, meaning you can define how they appear on the screen. Other containers exist only to ensure that their children are grouped; they do not have a visual display of their own.

Container Types	
Container	Description
Group	The simplest type of container in Flex. A Group can be used to contain children, but it does not have any visual appearance of its own.
SkinnableContainer	A container with all the same functionality as the Group, but also the ability to define its visual appearance on the screen.
BorderContainer	A type of SkinnableContainer that can be used to quickly surround children of a container with a border.
Panel	A type of SkinnableContainer, surrounded by a border, that can have a header and a control area called a control bar.
Application	A type of SkinnableContainer that is used as the root of your Flex application. Like the Panel, it can also have a control bar.

There are several more Flex containers, including `DataGroup` and `SkinnableDataContainer`, in addition to several specialized containers, such as `Form`, which will be used in the coming lessons. However, those containers follow a slightly different layout metaphor, so they will be introduced a bit later when their specific use can be explained more clearly.

Understanding Layout Objects

Layout objects work with containers (and other types of objects, as you will learn in later lessons) to determine how the grouped items of a container should appear on the screen. Flex provides a number of layout objects by default and allows you to create your own layout objects for complete customization.

Layout Object Types	
Layout Object	Description
<code>BasicLayout</code>	<code>BasicLayout</code> allows for absolute positioning. When using <code>BasicLayout</code> , you must note the specific <i>x</i> and <i>y</i> positions of each layout element.
<code>HorizontalLayout</code>	<code>HorizontalLayout</code> arranges children in a row, with each child positioned to the right of the previous one.
<code>VerticalLayout</code>	<code>VerticalLayout</code> arranges children in a column, with each child positioned below the previous one.
<code>ConstraintLayout</code>	<code>ConstraintLayout</code> arranges children based on defined columns (<code>ConstraintColumn</code>) and rows (<code>ConstraintRows</code>).
<code>FormLayout</code>	<code>FormLayout</code> is the default layout for Flex Forms and is composed of child <code>FormItemLayout</code> instances.
<code>TileLayout</code>	<code>TileLayout</code> arranges children in new rows and columns as necessary. You can specify whether items proceed horizontally or vertically before beginning a row or column.

Combining Containers and Layout Objects

Once you have chosen a container and a layout object, you assemble them in MXML to produce the desired effect. Look at the following examples of setting a layout object using the layout property to control the positioning of the buttons.

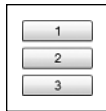
```
<s:Group>
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>

  <s:Button label="1"/>
  <s:Button label="2"/>
  <s:Button label="3"/>
</s:Group>
```



```
<s:Group>
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:Button label="1"/>
  <s:Button label="2"/>
  <s:Button label="3"/>
</s:Group>
```



If you do not specify a layout object, `BasicLayout` is used, meaning you must specify x and y positions for each button or they will all default to appear at the origin coordinates (0,0).

Scrolling Content

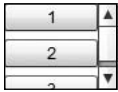
You will occasionally find a situation in an application in which it is desirable to scroll the contents of a group. In previous versions of Flex, every container had this functionality by default. While extremely convenient for the developer, it also meant that every container was burdened with this extra code even though it was hidden the vast majority of times. In the latest versions of Flex, you need to specifically indicate when an area is scrollable. This is accomplished via a special tag named `Scroller` that wraps your `Group` tag.

```

<s:Scroller height="65">
  <s:Group>
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>

    <s:Button label="1"/>
    <s:Button label="2"/>
    <s:Button label="3"/>
  </s:Group>
</s:Scroller>

```



Just wrapping the Group in a Scroller will not necessarily make a scroll bar appear. The Scroller will add scroll bars (vertical, horizontal, or both) as needed when there is not enough room to display the Group at full size. In the previous example, the height of the Scroller is specifically set to 65 pixels to ensure that a vertical scroll bar appears. If you do not set specific width and heights, then Flex will always try to fit the whole Group on the screen first and will resort to scrolling only if that is not possible.

Decoding MXML Tags

Before you begin the exercise in the next section, there is an important concept to learn. It is the difference between *class instances* and *properties* in MXML. If you look at the code snippet from the previous section, you will see a Flex button defined in MXML. Right now the `label` property of that Button is defined as an attribute of the Button's XML tag:

```
<s:Button label="3"/>
```

However, in MXML, you are also allowed to define this same property using child tags. In that case, the code would appear as follows:

```

<s:Button>
  <s:Label>3</s:Label>
</s:Button>

```

These two ways of defining the classes will yield identical results on the screen. After you have used Flex for a while, it will become a natural part of your development process to choose the correct syntax in a given situation; however, it can be very confusing when you are new to the language.

Now, how do you know, regardless of the definition style, which is a property and which is a class? The key to decoding this logic is to watch the case of the first letter after the namespace (after *s*: in this example). When the first letter is uppercase, such as the *B* in *Button*, the code is referring to a new instance of a class. When the first letter is lowercase, such as the *l* in *label*, you are setting a property of an instance of a class.

If you consider a slightly larger example from the previous code:

```
<s:Group>
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <s:Button label="1"/>
  <s:Button label="2"/>
  <s:Button>
    <s:label>3</s:label>
  </s:Button>
</s:Group>
```

The *G* in the `<s:Group>` tag is uppercase, so it refers to an instance of a Flex class named `Group`. The *l* in the `<s:layout>` tag is lowercase, so it is a property of the `Group`. The *V* in the `<s:VerticalLayout>` tag is uppercase, so it is referring to a new instance of a Flex class named `VerticalLayout`.

If you were to translate the code into words, it would read as follows: Create an instance of the `Group` class. Set the `layout` property of that `Group` instance to a new instance of the `VerticalLayout` class. Add three `Buttons` to that `Group` with the labels 1, 2, and 3, respectively.

Make sure this section makes complete sense before continuing in the book. If you ensure you understand these points, the rest of this lesson will be smooth sailing. If you are unsure, the remainder can be a very disheartening experience.

Laying Out the E-Commerce Application

The e-commerce application of `FlexGrocer` is the means through which customers shop for groceries. The top region of the application's user interface displays the store logo as well as navigation links that appear throughout the application. Below that is a series of clickable icons that users can use to browse the various categories of groceries (dairy, meat, fruit, and so on). Below the icons is an area for displaying products.

In this lesson, you will use both Design view and Source view to begin laying out the application. Design view is a powerful feature of Flash Builder but can be a very frustrating experience, especially when you are new to it. It is often very difficult to get objects to align correctly or to be placed inside the intended container on the screen. Therefore, you'll also see a code sample for everything you do in Design view. If your interface does not look like the one in the book as you proceed, feel free to switch to Source view and make the changes before switching back to Design view.

Starting Your Layout in Source View

The first steps of laying out your new application will be done in Source view as you define the area of your application that will hold the logo and some navigation elements.

- 1 Open the FlexGrocer.mxml file that you created in the previous lesson.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson03/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Ensure that Flash Builder is in Source view.

To switch between Design view and Source view in Flash Builder, click the buttons in the title bar near the top of the window.

- 3 Find and delete the Label tag with the text "My First Flex Application" that you added in the last lesson.
- 4 Insert a new controlBarLayout tag pair in place of the Label tag you just removed.

```
<s:controlBarLayout>  
</s:controlBarLayout>
```

This tag starts with a lowercase letter, indicating that it is a property of the Application object.

A *control bar* is a section of a container that is visually distinctive. In this application, you are going to use the control bar to hold a logo and some buttons for navigation.

- 5 Immediately inside the controlBarLayout tag pair, place a self-closing <s:BasicLayout> tag.

```
<s:controlBarLayout>  
  <s:BasicLayout/>  
</s:controlBarLayout>
```

Remember, a self-closing tag simply means that instead of having an open tag and a close tag like the controlBarLayout, you have just a single tag that ends in a forward slash and a greater than sign (</>).

Adding the `<s:BasicLayout/>` tag tells the Application that you want to use absolute positioning inside the control bar for this application. In other words, you will take responsibility for positioning the *x*- and *y*-coordinates of the items inside it.

- 6 Directly below the `controlBarLayout` tag pair, add a new tag pair named `<s:controlBarContent>`.

Inside this tag, you will define which items should appear in the control bar.

- 7 Add a `Button` tag inside the `controlBarContent` tag pair and set its `label` property to **Flex Grocer**.

```
<s:Button label="Flex Grocer"/>
```

Setting the `label` property of this `Button` will make it display “Flex Grocer” on the screen. Because you added this `Button` inside the `controlBarContent` tag pair, the `Button` will appear in the control bar area of the application.

Ensure that your code looks like the following sample before continuing:

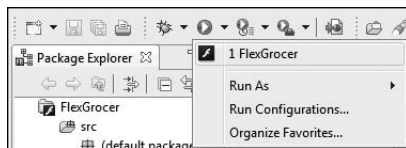
```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  minWidth="955" minHeight="600">
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>

  <s:controlBarLayout>
    <s:BasicLayout/>
  </s:controlBarLayout>

  <s:controlBarContent>
    <s:Button label="Flex Grocer"/>
  </s:controlBarContent>

</s:Application>
```

- 8 After verifying that your code looks like the example, save the `FlexGrocer.mxml` file and make sure you do not have any errors in the Problems view.
- 9 Choose `FlexGrocer` from the Run menu to execute your application in the web browser.



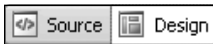
When your application launches, you should see a gray block near the top of the screen. This is the control bar. Inside that control bar you should see a single button with the words “Flex Grocer”. While the application may not do much yet, you have actually defined several properties, used a layout object, and added a child object to a container. It will get easier and faster from here. When you are finished admiring your work, close the web browser and get ready to learn about Design view.



Continuing Your Layout in Design View

You have already defined a portion of your application layout using MXML, but you will now use Design view to add several more elements and to define their properties.

- 1 Switch Flash Builder to Design view.

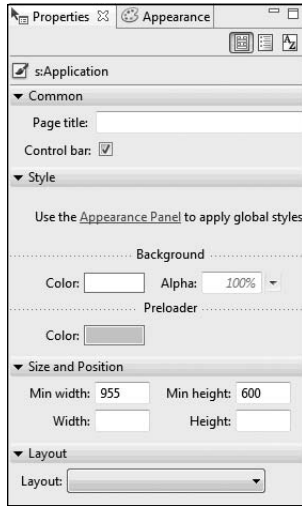


To switch between Design view and Source view in Flash Builder, click the buttons in the title bar at the top of the window. You will see a visual representation of your work so far.

- 2 Start by clicking anywhere in the large white background area of the screen.

The Properties panel on the right side of the screen should change so that *s:Application* is the heading. This is where you will set all component properties while in Design view.

- * **NOTE:** If the Properties panel is not currently visible on your screen, choose Window > Perspective > Reset Perspective (or Window > Reset Perspective on MacOS). This will reset your Design view options to the default settings and display the Properties panel.



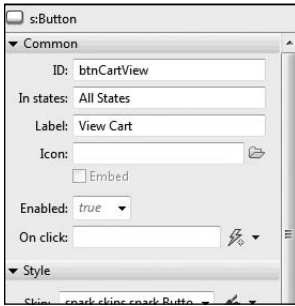
- 3 Click the Flex Grocer Button instance that you positioned in the previous exercise.

When you click the Button, the Properties panel on the right side of the screen will change to read *s:Button*, indicating that the Button you just selected is now being configured.

- 4 Toward the bottom of the Properties view, expand the Size and Position category by clicking the triangle next to the word *Size* (you may need to scroll down depending on your screen resolution). There are text boxes for Width, Height, X, and Y. Use the X and Y fields to set the *x*-coordinate to 5 and the *y*-coordinate to 5.

When you change the *y*-coordinate, the control bar will grow to accommodate the position of the Button. Later in the book, you will apply styles to set the company logo colors and size. For now, you are just placing the Button in the appropriate position. This is an example of using absolute-coordinates positioning.

- 5 Find the Components view; by default this will be in the lower-left corner of your screen. Open the Controls folder by clicking the triangle next to the word *Controls*, and drag a Button control to the control bar so the Button control is positioned near the right edge of the control bar. In the Properties view, give the Button control the ID **btnCartView** and the label **View Cart**.



- ▶ **TIP:** A blue bar will appear, indicating where other components exist horizontally or vertically from your position. This line will aid you in quickly lining up multiple components.

Don't worry about the exact x and y placement. Later in this lesson, you will learn how to use a constraint-based layout to position the button so that its right edge is always 10 pixels from the right edge of the Application object.

- 6 Drag a second Button control to the control bar, just to the left of the first Button control. In the Properties view, give the new Button control the ID **btnCheckout** and the label **Checkout**.

FlexGrocer users will click this button to indicate that they are finished shopping and want to complete the purchase of the selected products. Again, the exact placement will happen later in this lesson, when you learn about constraint-based layout.

- 7 Drag a Label control from the Controls folder and place it near the bottom-right edge of the screen. Double-click the Label and set the text property to (c) 2011, **FlexGrocer**.

Much like the Button controls you just added, you needn't worry about the exact placement of the Label control because it will be handled later with constraints.

- 8 In the Components panel, collapse the Controls folder and expand the Layout folder.
- 9 Drag a Group container from the Layout folder of the Components view and place it in the large white area below the control bar. Use the Properties panel to set the ID of the Group to **bodyGroup**. Then set both the height and width properties to **100%** and the x - and y -coordinates to **0**.
- 10 With the **bodyGroup** still selected, scroll to the bottom of the Properties panel. You will see a Layout drop-down menu. Choose `spark.layouts.HorizontalLayout`, indicating that you would like this Group to arrange its children horizontally.

This Group will hold the product details and shopping cart for the application. Remember that a Group with a `HorizontalLayout` displays its children horizontally. You will have products shown on the left and the shopping cart displayed on the right.

- 11 Drag another Group container from the Layout folder of the Components view and drop it inside the existing Group that you named bodyGroup. In the Properties view, give this new Group the ID **products** and then assign a height of **150** and width of **100%**.
- 12 At the bottom of the Properties panel, assign the new Group a spark.layouts.VerticalLayout, indicating that you would like this Group to arrange its children vertically.

This vertical group will hold the details for a product.

- 13 Before continuing, switch to Source view and ensure that your code matches the following code. If any tags are different or missing, fix them before continuing. It is okay if your code has slightly different values for the x and y properties of the Checkout Button, View Cart Button, and Label, as you have not set those yet.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               minWidth="955" minHeight="600">
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>

  <s:controlBarLayout>
    <s:BasicLayout />
  </s:controlBarLayout>

  <s:controlBarContent>
    <s:Button label="Flex Grocer" x="5" y="5" />
    <s:Button id="btnCartView" x="463" y="10" label="View Cart"/>
    <s:Button id="btnCheckout" x="541" y="10" label="Checkout"/>
  </s:controlBarContent>

  <s:Label x="500" y="546" text="(c) 2011, FlexGrocer"/>
  <s:Group id="bodyGroup" x="0" y="0" width="100%" height="100%">
    <s:layout>
      <s:HorizontalLayout/>
    </s:layout>
    <s:Group id="products" width="100%" height="150">
      <s:layout>
        <s:VerticalLayout/>
      </s:layout>
    </s:Group>
  </s:Group>

</s:Application>
```

Defining the Product Section

Once you verify that your source code matches the example code, switch back to Design view, where you will continue defining the product section. Now, you will begin defining the controls that will eventually represent all the products in your online grocery store.

▶ **TIP:** Sometimes when switching between Source and Design views, you can lose track of the Flash Builder Properties panel in Design view. If this panel ever goes missing, simply choose Windows > Properties (Window > Show View > Properties on Mac OS) to bring it back.

- 1 Drag a Label control from the Controls folder of the Components view to the vertical Group to which you assigned the ID of products in the previous exercise. When looking at Design view, this vertical group will have a faint border starting directly below the control bar and continuing down for 150 pixels, where it crosses the screen. You can drop the Label anywhere in this area.
- 2 Set the ID of the Label control to **prodName** and the Text property to **Milk**.
- 3 Drag a second Label control below the first one. Give the second one the ID **price** and set **\$1.99** as the Text.

Because these new Label controls are children of the Group container, and the Group has a VerticalLayout object, the product name appears above the price of the product.

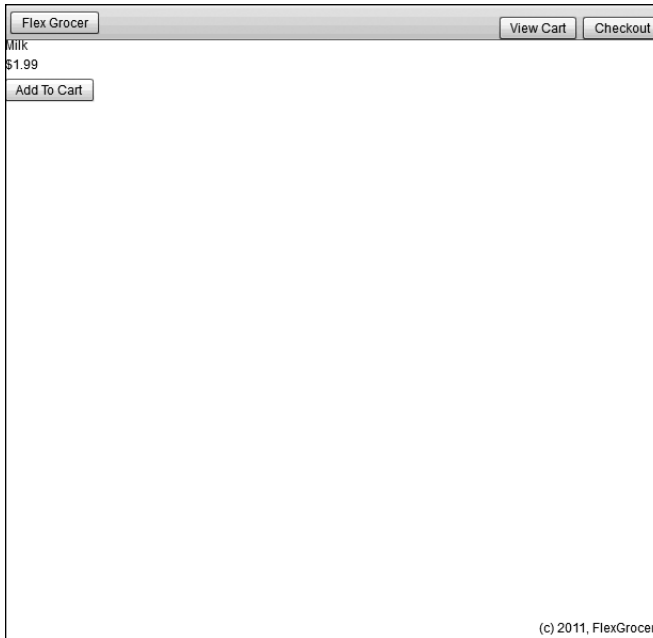
▶ **TIP:** If you open Outline view by clicking the Outline tab (this is adjacent to the Components tab you have been using so far), you can see the hierarchy of your application. The root is the Application tag, which contains a Label (copyright) and a Group named bodyGroup as children along with controlBarContent and a controlBarLayout as properties. You can also see the various children of the controlBarContent and the bodyGroup. If you expand the Group named products, you will see the two labels you just added. This is a useful view if you want to make a change to a component. But, it can be difficult to select just the products Group in Design view. Click Outline view to easily select the products Group.

- 4 Add a Button control below the two Label controls, with an ID of **add** and the label **Add To Cart**.

For each product, you want the name of the product and its price to be displayed. The add Button gives users the ability to add a product to their shopping cart. Because the two Label controls and the Button control are in a group with a vertical layout, they appear one above the other. You'll add functionality for the Button control in a later lesson.

- 5 Save the file and click Run.

You can clearly see the difference between elements in the control bar section and those in the body.



Working with Constraint-Based Layouts

Flex supports constraint-based layouts that let you arrange elements of the user interface with the freedom and pixel-point accuracy of absolute positioning while being able to set constraints to stretch or move the components when the user resizes the window. This method of controlling the size and position of components is different from laying out nested containers (like the Group containers in the previous exercise).

In constraint-based layouts, all the controls are positioned in relation to the edges of a parent container, which has been set with a `BasicLayout` to allow absolute positioning. With the exception of some specialized containers such as `Form` (which you will use in subsequent lessons), you can use the `BasicLayout` on any `Group` or `SkinnableContainer`, including `Application` and `Panel`.

Containers using a `BasicLayout` object require that elements be positioned to absolute coordinates; however, layout constraints allow users to dynamically adjust the layout based on

the window size of their browsers. For example, if you want a label to always appear in the bottom-right corner of an application regardless of the browser size, you can anchor the control to the right edge of the parent container. The control's position is then always maintained relative to the right edge of the parent container.

In Flex, this is accomplished via layout anchors. They are used to specify how a control should appear relative to the edge of the parent container. To ensure that a control is a certain distance from the bottom and right edges, you will select the check boxes below and to the right of the control in the Constraints area in the Layout section of the Properties view, and use the text boxes to specify the number of pixels away from the edge of the container where you want the control constrained.

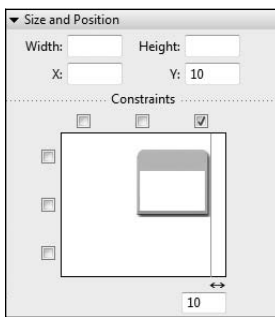
Flex allows constraints from the top, vertical center, bottom, left, horizontal center, or right of a container.

► **TIP:** All constraints are set relative to the edges of the container, as long as the container uses absolute positioning (BasicLayout). Constraints cannot be set relative to other controls or containers.

- 1 Open the FlexGrocer.mxml file that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer-PreConstraints.fxp project from the Lesson03/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Ensure that you are in Design view, then find and select the Checkout Button. Toward the bottom of the Properties view, in the Constraints area of the Size and Position section, add a constraint so the right edge of the Button is **10** pixels away from the right edge of the container. Make sure that the Y position of this control is set to **10** pixels.



To set a constraint from the right edge, click the rightmost check box above the button icon in the Constraints area. In the text box that appears, enter the number of pixels away from the edge you want the button to be. If the label seems to disappear from the screen, check the scroll bars on the bottom of Design view. By default, Design view shows you just a portion of the application, and you may need to scroll occasionally to find what you need.

- 3** Find and select the View Cart Button. Add a constraint so that the right edge of the button is **90** pixels from the right edge of the container. Make sure that the Y position of this control is set to **10** pixels.

You now have it set so that, regardless of the width of the browser, the two navigation buttons are always anchored relative to the top-right edge of the container.

- 4** Find and select the Label control (outline view may be very helpful if you have trouble selecting this otherwise) with the copyright notice. Constrain this Label so that it is 10 pixels above the bottom and 10 pixels away from the right edge of its container. Click the check box in the top-right corner of the Constraints area, and enter **10** in the text box that appears. Also, click the bottom check box and enter **10** in the text box that appears.

Because the copyright label is below other containers, it is probably easiest to select it using the Outline view. These settings ensure that, regardless of the width of the Label control, its bottom-right edge will always be 10 pixels above and 10 pixels to the left of the bottom-right corner of the application.

If you switch to Source view, the entire file should look similar to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    minWidth="955" minHeight="600">
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>

    <s:controlBarLayout>
        <s:BasicLayout />
    </s:controlBarLayout>

    <s:controlBarContent>
        <s:Button label="Flex Grocer" x="5" y="5" />
        <s:Button id="btnCartView" y="10" right="90" label="View Cart"/>
        <s:Button id="btnCheckout" y="10" right="10" label="Checkout"/>
    </s:controlBarContent>

    <s:Label right="10" bottom="10" text="(c) 2011, FlexGrocer"/>
    <s:Group id="bodyGroup" x="0" y="0" width="100%" height="100%">
```

```

<s:layout>
  <s:HorizontalLayout/>
</s:layout>
<s:Group id="products" width="100%" height="150">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <s:Label id="prodName" text="Milk"/>
  <s:Label id="price" text="$1.99"/>
  <s:Button id="add" label="Add To Cart"/>
</s:Group>
</s:Group>
</s:Application>

```

Your code may differ slightly, depending on the order in which you added the items and set properties. Don't worry; the order isn't particularly important in this case. Every container and control that you added in Design view is represented by a tag in Source view. When you add elements inside a container, they appear as child tags to the container's tag. Also note that the layout constraints are set as attributes of the related component.

- 5 Switch back to Design view and insert a second Group container in the bodyGroup container, to the right of the products (the bodyGroup is the first container you added whose width and height are set to 100%). Set the ID of the new Group to **cartGroup**, clear the Width property so it is blank, and set the Height to **100%**. Remember, you can always choose the bodyGroup from the Outline view if you have difficulty finding it.

If you accidentally place the new Group in the wrong container, the easiest fix is to switch to Source view and move the tags yourself. The code in Source view for this area should look like this:

```

<s:Group id="bodyGroup" x="0" y="0" width="100%" height="100%">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <s:Group id="products" width="100%" height="150">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <s:Label id="prodName" text="Milk"/>
    <s:Label id="price" text="$1.99"/>
    <s:Button id="add" label="Add To Cart"/>
  </s:Group>
  <s:Group id="cartGroup" height="100%">
  </s:Group>
</s:Group>

```

- 6** In Design view, set the layout of the cartGroup container to VerticalLayout.

If you can't find the cartGroup, remember to choose it from the Outline view and scroll in the Design view window until you see the highlighted container.

- 7** Add a Label control in the cartGroup container with the text property set to **Your Cart Total: \$0**.

To the right of the products, there will always be a summary of the shopping cart, indicating whether there are items in the cart and what the current subtotal is.

*** NOTE:** At this point you have set the products container to take 100% of the space, but then you added the cartGroup to its right and added a Label. Isn't that a problem, as you are now using more than 100%? Flex enables you to assign more than 100% total width or height for a container. Flex containers take this into account and divide the space proportionally based on the requested percentages. Because more space was requested than is available, each request receives a relative portion based on the available space. If any elements were assigned a fixed width (that is, a number of pixels instead of a percentage), the fixed size requests are subtracted from the available space before any relative size requests are allocated.

- 8** From the Controls folder of the Components view, drag a Button control below the newest Label control and set the label property of the Button to **View Cart**.

This Button shows users the full contents of their shopping cart.

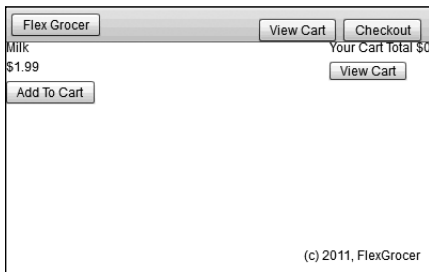
If you accidentally place any components in the wrong container, switch to Source view and move the tags yourself. The code in Source view for this area should look like this:

```
<s:Group id="bodyGroup" x="0" y="0" width="100%" height="100%">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <s:Group id="products" width="100%" height="150">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <s:Label id="prodName" text="Milk"/>
    <s:Label id="price" text="$1.99"/>
    <s:Button id="add" label="Add To Cart"/>
  </s:Group>
  <s:Group id="cartGroup" height="100%">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <s:Label text="Your Cart Total $0"/>
    <s:Button height="17" label="View Cart"/>
  </s:Group>
</s:Group>
```

- 9 In Outline view, choose the Application. In the Properties panel, remove the Min width and Min height values of 955 and 600.

As the application runs, you can resize the browser and see that the buttons and bottom text are always properly constrained. A minimum width and height would prevent resizing from occurring on smaller screens.

- 10 Save the file and click Run.



Working with View States

You can use Flash Builder to create applications that change their appearance based on the task the user is performing. For example, the e-commerce application starts by showing users the various products they can buy. When they start adding items to the cart, you want to add something to the view, such as the total cost, so users can get a feel for what is currently in the cart. Finally, users need a way to view and manage the full contents of the shopping cart.

Creating View States

In Flex, you can add this kind of interactivity with view states. A *view state* is one of several views that you define for an application or a custom component. Every MXML page has at least one state, referred to as the *default state*, which is represented by the default layout of the file.

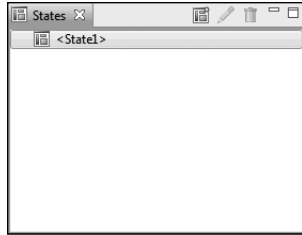
Additional states are represented in the MXML as modified versions of the base view state or of other states.

- 1 Open the FlexGrocer.mxml file that you used in the previous exercise.


Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer-PreStates.fxp project from the Lesson03/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Ensure you are in Design view. If it is not already open, open the States view in Flash Builder.

If you don't currently see the States view when you look at Flash Builder in Design view, you can add it to the view by choosing Window > States (Windows > Show View > States on Mac OS). Notice that there is already one state created to represent the default layout of the application.



- 3 Create a new state named **cartView**, which is a duplicate of <State1>.

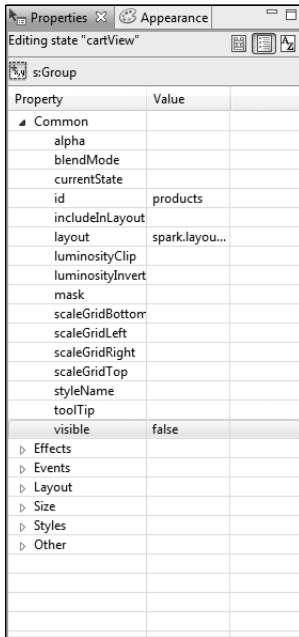
You can create a state by clicking the New State icon  at the top of the States view or by right-clicking in the view and selecting the New option. The cartView state will show users the details of all the items they have added to their cart. Leave the default options, and click OK.

- 4 With the new cartView state selected, click the products container and set its height and width to **0**, then choose the cartGroup container and set its height and width values to **100%**.

In the cartView state, the shopping cart will entirely replace the products in the center of the screen. Therefore, you will resize the products container to take up no space and resize the cartGroup container to take up all the available space.

At this point, the controls on your screen will be quite a mess. You will likely see an ugly combination of all the controls in the system on top of each other. This is a very important lesson. In Flex, the width and height properties are used to compute the location of items on the screen. In this case, you told Flex that the products container will not take any space, so Flex responded by moving the cartGroup container left to take the newly available space. However, not allocating space to an item on the screen does not ensure that it is invisible.

- 5 Select the products container and change its visible property to **false**. You can do this by clicking the CategoryView of the Properties panel, finding the visible property, and changing its value to **false**.



- ▶ **TIP:** It has been said several times so far in this lesson, but it is so important it is worth repeating: Use the Outline view to find containers when they are difficult to locate on the screen.
- 6 Ensure that the cartView state is still selected in States view and then drag a DataGrid control from the Controls folder of the Components view and drop it below the View Cart button. Set the ID of the DataGrid control to **dgCart**, and set the DataGrid control's width to **100%**.

In a later lesson, the DataGrid control will be used to show the user the full contents of the cart.

Make sure you are adding the DataGrid control to the cartGroup container. Your application and code will look a bit different if you accidentally add the DataGrid control before the cartGroup container.

If you look at the file in Source view, you should see that the DataGrid has been added to the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:states>
        <s:State name="State1"/>
        <s:State name="cartView"/>
    </s:states>
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>

    <s:controlBarLayout>
        <s:BasicLayout />
    </s:controlBarLayout>

    <s:controlBarContent>
        <s:Button label="Flex Grocer" x="5" y="5" />
        <s:Button id="btnCartView" y="10" right="90" label="View Cart"/>
        <s:Button id="btnCheckout" y="10" right="10" label="Checkout"/>
    </s:controlBarContent>

    <s:Label right="10" bottom="10" text="(c) 2011, FlexGrocer"/>
    <s:Group id="bodyGroup" x="0" y="0" width="100%" height="100%">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Group id="products" width="100%" height="150"
            visible.cartView="false" width.cartView="0" height.cartView="0">
            <s:layout>
                <s:VerticalLayout/>
            </s:layout>
            <s:Label id="prodName" text="Milk"/>
            <s:Label id="price" text="$1.99"/>
            <s:Button id="add" label="Add To Cart"/>
        </s:Group>
        <s:Group id="cartGroup" height="100%"
            width.cartView="100%">
            <s:layout>
                <s:VerticalLayout/>
            </s:layout>
            <s:Label text="Your Cart Total $0"/>
            <s:Button height="17" label="View Cart"/>
        </s:Group>
    </s:Group>
</s:Application>
```

```

<s:DataGrid includeIn="cartView" requestedRowCount="4">
  <s:columns>
    <s:ArrayList>
      <s:GridColumn dataField="dataField1"
        headerText="Column 1"></s:GridColumn>
      <s:GridColumn dataField="dataField2"
        headerText="Column 2"></s:GridColumn>
      <s:GridColumn dataField="dataField3"
        headerText="Column 3"></s:GridColumn>
    </s:ArrayList>
  </s:columns>
  <s:typicalItem>
    <fx:Object dataField1="Sample Data"
      dataField2="Sample Data"
      dataField3="Sample Data"></fx:Object>
  </s:typicalItem>
  <s:ArrayList>
    <fx:Object dataField1="data1"
      dataField2="data1"
      dataField3="data1"></fx:Object>
    <fx:Object dataField1="data2"
      dataField2="data2"
      dataField3="data2"></fx:Object>
    <fx:Object dataField1="data3"
      dataField2="data3"
      dataField3="data3"></fx:Object>
    <fx:Object dataField1="data4"
      dataField2="data4"
      dataField3="data4"></fx:Object>
  </s:ArrayList>
</s:DataGrid>
</s:Group>
</s:Group>

</s:Application>

```

7 Save the file.

Note some of the new syntax added during this operation. First, in the `DataGrid` class, you will see the `includeIn` property, indicating that this control should appear on the screen only when in the `cartView` state. Second, the `products` container still has a `width` of `100%` and `height` of `150`; however, it also has `width.cartView="0"` and `height.cartView="0"`. This syntax instructs Flex to set those properties in the corresponding states.

Testing the file now shouldn't show any differences because you haven't added any ability for the user to toggle between the states. In the next exercise, you will add that navigational ability.

Controlling View States

Each MXML component has a property called `currentState`. You can use this property to control which state of the application is shown to a user at any given time.

- 1 Open the `FlexGrocer.mxml` file that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer-PreControl.fxp` project from the `Lesson03/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Switch to Design view and, if it is not already open, open the States view in Flash Builder and select **State1** to set the current state.

You will add functionality to the default view state so that users can navigate to the other states of the application.

- 3 Switch the Properties view to the Standard View mode, and choose the View Cart Button control from the `cartGroup` container. In the Properties view, set the `on click:` property to `this.currentState='cartView'`.

Events such as the Button's `click` will be explored in detail in Lesson 5, "Handling Events." The important thing to understand now is that when the user clicks the link, the view will change to the `cartView` state.

▼ **CAUTION!** The state name is case sensitive and must exactly match the name as you typed it in the previous exercise. You should use single quotes around the state name when entering it in Design view; otherwise, Flash Builder will create escape code (`"`;) for the double quotes, and your code will be confusing to read.

- 4 Choose the View Cart Button control from the control bar. In the Properties view, also set the `on click:` property to `this.currentState='cartView'`. You now have two ways to enter the `cartView` state.

- 5 Switch to the `cartView` state. Add a new Button control below the `DataGrid` control with the label set to **Continue Shopping** and the `click` property set to `this.currentState=''`. Note this is a single open and single close quote, as opposed to one double quote.

Setting `currentState` to an empty string resets the application to its default state.

- 6 Delete the View Cart Button that is inside the `cartGroup` from the `cartView` state.

When the user is viewing the cart, there is no need for a View Cart Button. You can delete the Button by selecting it in Design view and pressing Delete.

The completed application as shown in Source view should read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:states>
        <s:State name="State1"/>
        <s:State name="cartView"/>
    </s:states>
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>

    <s:controlBarLayout>
        <s:BasicLayout />
    </s:controlBarLayout>

    <s:controlBarContent>
        <s:Button label="Flex Grocer" x="5" y="5" />
        <s:Button id="btnCartView" y="10" right="90" label="View Cart"
            click.State1="this.currentState='cartView'"/>
        <s:Button id="btnCheckout" y="10" right="10" label="Checkout"/>
    </s:controlBarContent>

    <s:Label right="10" bottom="10" text="(c) 2011, FlexGrocer"/>
    <s:Group id="bodyGroup" x="0" y="0" width="100%" height="100%">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:Group id="products" width="100%" height="150"
            visible.cartView="false" width.cartView="0" height.cartView="0">
            <s:layout>
                <s:VerticalLayout/>
            </s:layout>
            <s:Label id="prodName" text="Milk"/>
            <s:Label id="price" text="$1.99"/>
            <s:Button id="add" label="Add To Cart"/>
        </s:Group>
        <s:Group id="cartGroup" height="100%"
            width.cartView="100%">
            <s:layout>
                <s:VerticalLayout/>
            </s:layout>
            <s:Label text="Your Cart Total $0"/>
            <s:Button includeIn="State1" height="17" label="View Cart"
                click="this.currentState='cartView'"/>
            <s>DataGrid includeIn="cartView" requestedRowCount="4">
                <s:columns>
                    <s:ArrayList>
```

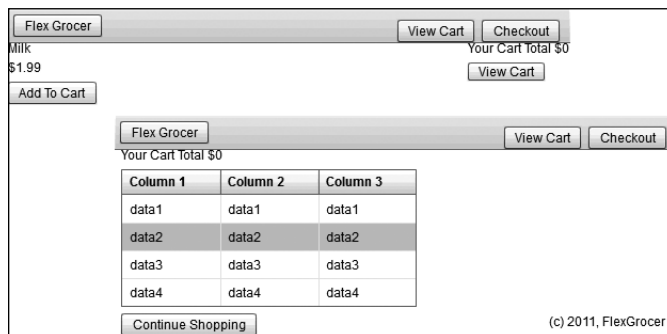
```

        <s:GridColumn dataField="dataField1"
            headerText="Column 1"></s:GridColumn>
        <s:GridColumn dataField="dataField2"
            headerText="Column 2"></s:GridColumn>
        <s:GridColumn dataField="dataField3"
            headerText="Column 3"></s:GridColumn>
    </s:ArrayList>
</s:columns>
<s:typicalItem>
    <fx:Object dataField1="Sample Data"
        dataField2="Sample Data"
        dataField3="Sample Data"></fx:Object>
</s:typicalItem>
<s:ArrayList>
    <fx:Object dataField1="data1"
        dataField2="data1"
        dataField3="data1"></fx:Object>
    <fx:Object dataField1="data2"
        dataField2="data2"
        dataField3="data2"></fx:Object>
    <fx:Object dataField1="data3"
        dataField2="data3"
        dataField3="data3"></fx:Object>
    <fx:Object dataField1="data4"
        dataField2="data4"
        dataField3="data4"></fx:Object>
</s:ArrayList>
</s:DataGrid>
<s:Button includeIn="cartView" label="Continue Shopping"
    click="this.currentState= ''"/>
</s:Group>
</s:Group>

</s:Application>

```

- 7** Save and test the application. You can now navigate between the states by clicking the buttons to which you added code.



Refactoring

Refactoring is one of the least understood and most useful tools in a developer's arsenal. It is particularly relevant for a Flex and ActionScript developer, because dynamic interfaces are often recombined with code during the prototype and development stages of a project.

Refactoring is simply the process of reorganizing your code in a way that is better suited to a long-term goal without changing the way it functions. Long-term goals might include increasing the maintainability of the software, modifying the architecture to make additional development steps easier, or simply changing the location and structure of the project to make it more understandable to a new developer. However, one thing is always true: At the end of a successful refactoring session, the changes will be imperceptible to an individual running the application who does not look at the source code. The application functions the same way.

Many developers find this notion and this process frustrating. Why would you want to spend time changing code you have already written if it makes no noticeable change in the application's execution? The answers are varied, but here are a few important ones.

- **Learning:** Learning a new language and continuing to use it is an ongoing experience. You will be learning new things and techniques every day. That often leads to the realization that the code you wrote days, weeks, or months ago may be inefficient, or even ineffective in certain circumstances. Keeping a keen eye on what you have written in the past and being willing to revisit it often provides a more cohesive code base and tighter, more maintainable code.
- **Duplication and redundancy:** As you are developing, it is extremely common to need the same functionality in multiple places in your application. Usually due to time constraints, this code stays forever duplicated. One of the many problems with this is that later, when that code needs to change, you have to be sure to hunt down and fix all the places it is used. A willingness to avoid duplication and move shared code into new places as you continue developing can not only eliminate large headaches down the road, but can also makes your day-to-day development more efficient and faster.
- **The big picture:** Often it is difficult, if not impossible, to know how all the pieces in a system will fit together when you begin writing code. If these pieces written early in the process are set in stone, you will end up bending or breaking code down the line when you try to integrate. If you are comfortable with the idea of refactoring your code as needed throughout the process, you can hone your vision of the system as you progress, ending up with objects and structures that work more cohesively when they're finished.

We have a couple of reasons for addressing refactoring here. First, many new developers to the Flex and ActionScript world attempt to apply a rigid approach to their development that does not include refactoring. Over the course of time, we have noticed that these developers, above all others, struggle against the language instead of learning to use it as a tool. We simply want you to avoid that pain.

Second, throughout this book you are going to be learning. In fact, quite often you are going to learn multiple techniques for accomplishing the same goal. It is not always feasible to introduce the one “right” way to do something from the beginning because these concepts tend to build upon each other. So, once you have learned enough to approach writing something in a new way, you will end up refactoring it. This provides two benefits: the ability to understand multiple ways to accomplish a goal (and hopefully the reasons why you would or would not use one or the other) and the ability to hone the code base into a final application with valid examples for reference.

That said, you are going to refactor the application you have built to date, to make it easier to maintain as you continue through this book.

Using Composed Containers

As you learned in this lesson, most containers in Flex accept a layout object that dictates the orientation of their children. This layout object is generally specified by adding a `LayoutObject` to a `Group` using the `layout` property, as the following example shows:

```
<s:Group>
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>

  <s:Button label="1"/>
  <s:Button label="2"/>
  <s:Button label="3"/>
</s:Group>
```

While this provides the utmost flexibility, it does require a little extra typing each time you create a new `Group`. In a small application this is not a very big issue; however, in a large application, adding layout objects to tens of thousands of `Groups` can become tedious. To solve this problem, Flex allows you to create new components composed of existing components and properties. You can then use these new constructs as shortcuts to desired functionality.

The Flex framework has prebuilt a few of these shortcuts in the form of two containers called `VGroup` and `HGroup`. In the following chart, the horizontal columns are functionally equivalent.

Alternative Shortcuts

Using layout Property	Composed Version
<pre><s:Group> <s:layout> <s:HorizontalLayout/> </s:layout> <s:Button label="1"/> <s:Button label="2"/> <s:Button label="3"/> </s:Group></pre>	<pre><s:HGroup> <s:Button label="1"/> <s:Button label="2"/> <s:Button label="3"/> </s:HGroup></pre>
<pre><s:Group> </s:layout> <s:VerticalLayout/> </s:layout> <s:Button label="1"/> <s:Button label="2"/> <s:Button label="3"/> </s:Group></pre>	<pre><s:VGroup> <s:Button label="1"/> <s:Button label="2"/> <s:Button label="3"/> </s:VGroup></pre>

If you were to examine the VGroup and HGroup source code, you would find that they are little more than the Group you have already learned to use with the layout property preset for your use. In Lessons 9, “Breaking the Application into Components,” and Lesson 18, “Creating Custom ActionScript Components,” you will learn to create your own components wherever you see a similar opportunity to reuse code.

Refactoring Your Application

In this section, you will convert all the Groups with HorizontalLayouts to HGroups, and Groups with VerticalLayouts to VGroups. The goal of this exercise is to successfully change the internal structure of the application without changing its functionality.

- 1 Open the FlexGrocer.mxml file that you used in the previous exercise.

Alternatively, if you didn’t complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer-PreRefactor.fxp project from the Lesson03/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Switch to Source view.
- 3 Find the group named `bodyGroup` and change it to an `HGroup`. Be sure to also change the closing tag for this group.
- 4 Eliminate the tag for the `layout` property and the `HorizontalLayout` object from within the `bodyGroup`.
- 5 Find the `products` group and change it to a `VGroup`. Be sure to change the closing tag as well.
- 6 Eliminate the tag for the `layout` property and the `VerticalLayout` object from within the `products` group.
- 7 Repeat this process for the `cartGroup`.

When you are finished refactoring the application, your code should appear as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:states>
        <s:State name="State1"/>
        <s:State name="cartView"/>
    </s:states>
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>

    <s:controlBarLayout>
        <s:BasicLayout />
    </s:controlBarLayout>

    <s:controlBarContent>
        <s:Button label="Flex Grocer" x="5" y="5" />
        <s:Button id="btnCartView" y="10" right="90" label="View Cart"
            click.State1="this.currentState='cartView'"/>
        <s:Button id="btnCheckout" y="10" right="10" label="Checkout"/>
    </s:controlBarContent>

    <s:Label right="10" bottom="10" text="(c) 2011, FlexGrocer"/>
    <s:HGroup id="bodyGroup" x="0" y="0" width="100%" height="100%">
        <s:VGroup id="products" width="100%" height="150"
            visible.cartView="false" width.cartView="0" height.cartView="0">
            <s:Label id="prodName" text="Milk"/>
            <s:Label id="price" text="$1.99"/>
            <s:Button id="add" label="Add To Cart"/>
        </s:VGroup>
        <s:VGroup id="cartGroup" height="100%"
```

```

        width.cartView="100%")
<s:Label text="Your Cart Total $0"/>
<s:Button includeIn="State1" height="17" label="View Cart"
        click="this.currentState='cartView'"/>
<s:DataGrid includeIn="cartView" requestedRowCount="4">
    <s:columns>
        <s:ArrayList>
            <s:GridColumn dataField="dataField1"
                headerText="Column 1"/></s:GridColumn>
            <s:GridColumn dataField="dataField2"
                headerText="Column 2"/></s:GridColumn>
            <s:GridColumn dataField="dataField3"
                headerText="Column 3"/></s:GridColumn>
        </s:ArrayList>
    </s:columns>
    <s:typicalItem>
        <fx:Object dataField1="Sample Data"
            dataField2="Sample Data"
            dataField3="Sample Data"/></fx:Object>
    </s:typicalItem>
    <s:ArrayList>
        <fx:Object dataField1="data1"
            dataField2="data1"
            dataField3="data1"/></fx:Object>
        <fx:Object dataField1="data2"
            dataField2="data2"
            dataField3="data2"/></fx:Object>
        <fx:Object dataField1="data3"
            dataField2="data3"
            dataField3="data3"/></fx:Object>
        <fx:Object dataField1="data4"
            dataField2="data4"
            dataField3="data4"/></fx:Object>
    </s:ArrayList>
</s>DataGrid>
<s:Button includeIn="cartView" label="Continue Shopping"
        click="this.currentState"/>
</s:VGroup>
</s:HGroup>

</s:Application>

```

8 Save the file and click Run.

You should have the same functionality with the View Cart Button as before and see absolutely no change in functionality, yet have slightly more maintainable code.

What You Have Learned

In this lesson, you have:

- Used containers and layout objects (pages 33–50)
- Begun an application layout in Source view (pages 51–53)
- Laid out an application in Design view (pages 53–58)
- Worked with constraint-based layouts (pages 58–63)
- Worked with view states (pages 63–67)
- Controlled view states (pages 68–70)
- Refactored your application (pages 71–75)

This page intentionally left blank

LESSON 4

What You Will Learn

In this lesson, you will:

- Define the user interface (UI) for the e-commerce FlexGrocer application
- Use simple controls such as the Image control, text controls, and CheckBox control
- Define the UI for the checkout screens
- Use the Form container to lay out simple controls
- Use data binding to connect controls to a data model

Approximate Time

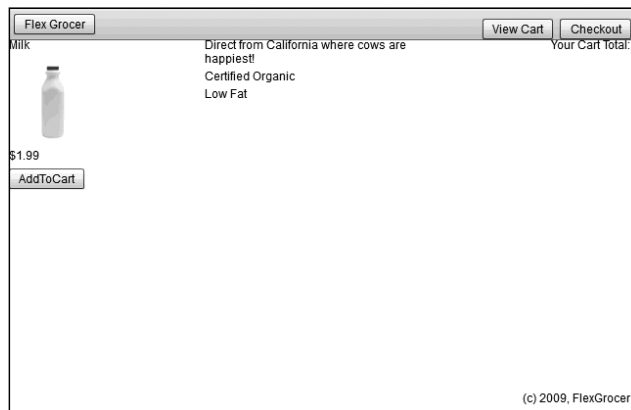
This lesson takes approximately 45 minutes to complete.

LESSON 4

Using Simple Controls

In this lesson, you will add user interface elements to enable the customer to find more details about the grocery items and begin the checkout process. An important part of any application is the user interface, and Adobe Flex contains elements such as buttons, text fields, and radio buttons that make building interfaces easier. Simple controls can display text and images and also gather information from users. You can tie simple controls to an underlying data structure, and they will reflect changes in that data structure in real time through data binding. You're ready to start learning about the APIs (application programming interfaces) of specific controls, which are available in both MXML and ActionScript. The APIs are fully documented in the ActionScript Language Reference, often referred to as ASDoc, which is available at http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html.

The Flex framework has many tools that make laying out simple controls easier. All controls are placed within containers (see Lesson 3, "Laying Out the Interface"). In this lesson, you'll become familiar with simple controls by building the basic user interface of the application that you will develop throughout this book. You'll also learn about timesaving functionality built into the framework, such as data binding and capabilities of the Form layout container.



FlexGrocer with Image and text controls bound to a data structure

Introducing Simple Controls

Simple controls are provided as part of the Flex framework and help make rich Internet application development easy. Using controls, you can define the look and feel of your buttons, text, combo boxes, and much more. Later in this book, you'll learn how to customize controls to create your own unique look and feel. Controls provide a standards-based methodology that makes learning how to use them easy. Controls are the foundation of any RIA.

The Flex SDK includes an extensive class library for both simple and complex controls. All these classes can be instantiated via an MXML tag or as a standard ActionScript class, and their APIs are accessible in both MXML and ActionScript. The class hierarchy comprises nonvisual classes as well, such as those that define the event model, and it includes the display attributes that all simple controls share.

You place the visual components of your Flex application inside containers, which establish the size and positioning of text, controls, images, and other media elements (you learned about containers in the previous lesson). All simple controls have events that can be used to respond to user actions, such as clicking a button, or system events, such as another component being drawn (events will be covered in detail in the next lesson). You will learn in later lessons how to build your own events. Fundamentally, events are used to build easily maintainable applications that reduce the risk that a change to one portion of the application will force a change in another. This is often referred to as building a “loosely coupled” application.

Most applications need to display some sort of text, whether it be static or dynamically driven from an outside source like an XML file or a database. Flex has a number of text controls that can be used to display editable or noneditable text:

- **Label:** You have already used the Label control to display text. The Label control cannot be edited by an end user; if you need that functionality, you can use a TextInput control.
- **TextInput:** The TextInput control is used for data input. It is limited to a single line of text.
- **RichText:** The RichText control is used to display multiple lines of text, but is not editable and does not display scroll bars if the text exceeds the available screen space.
- **TextArea:** The TextArea component is useful for displaying multiple lines of text, either editable or noneditable, with scroll bars if the available text exceeds the available screen space.

All text controls support HTML 1.0 and a variety of text and font styles.

*** NOTE:** All four text controls mentioned here support Adobe's Flash Text Engine and some of the controls (RichText and RichEditableText) support even more advanced layout using the Text Layout Framework (TLF). While you will not be using TLF as part of the application in this book, many new and interesting features are available with TLF. You can learn about TLF on Adobe's open source site: <http://opensource.adobe.com/wiki/display/tlf/Text+Layout+Framework>.

To populate text fields at runtime, you must assign an ID to the control. Once you have done that, you can access the control's properties; for example, all the text controls previously mentioned have a `text` property. This property enables you to populate the control with plain text using either an ActionScript function or inline data binding. The following code demonstrates assigning an ID to the label, which enables you to reference the Label control in ActionScript:

```
<s:Label id="myLabel"/>
```

You can populate any text control at runtime using data binding, which is denoted by curly bracket syntax in MXML. The following code will cause the `yourLabel` control to display the same text as the `myLabel` control in the previous example:

```
<s:Label id="yourLabel" text="{myLabel.text}"/>
```

Also, you can use data binding to bind a simple control to underlying data structures. For example, if you have XML data, which might come from a server-side dataset, you can use data binding to connect a simple control to the data structure. When the underlying data changes, the controls are automatically updated to reflect the new data. This provides a powerful tool for the application developer.

The Flex framework also provides a powerful container for building the forms that we will cover in this lesson. The Form container allows developers to create efficient, good-looking forms with minimal effort. Flex handles the heading, spacing, and arrangement of form items automatically.

Displaying Images

In this exercise, you will display images of grocery products. To do this, you must use the Image control to load images dynamically. The Image control can load JPG, GIF, SWF, and PNG files at runtime. If you are developing an offline application that will not access the Internet, you can use the `@Embed` directive to include the Image control in the completed SWF file.

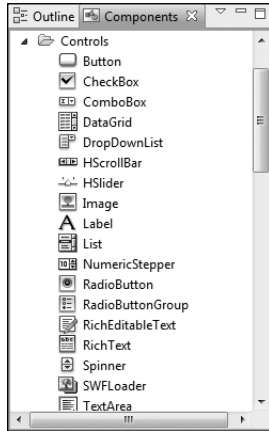
- 1 Open the `FlexGrocer.mxml` file that you created in the previous lesson.

If you didn't complete the previous lesson, you can import the `Lesson04/start` files. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2** Switch Flash Builder to Design view by clicking the Design View button.



- 3** Be sure that the Components view is open. If it's not, choose Window > Components.

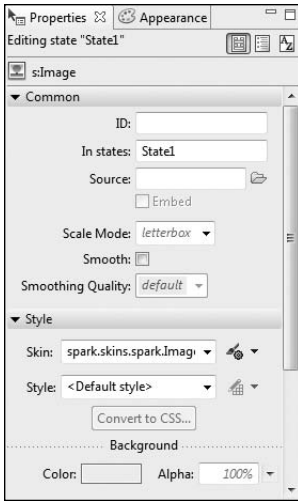


- 4** Select the Image control from the Controls folder and drag the control between the Milk and 1.99 Label controls you already added.



When you drag the Image control from the Components view to the container, Flash Builder automatically adds the MXML to place the Image control on the screen and positions it where you drop it.

- 5** Be sure that the Flex Properties view is open. If it's not, choose Window > Properties.



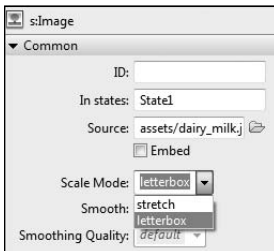
The Flex Properties view shows important attributes of the selected component—in this case, the Image control. You can see the Source property, which specifies the path to the Image file. The ID of the Image control references the instance created from the `<s:Image>` tag or Image class in ActionScript.

- 6 Click the Source folder icon and navigate to the assets directory. Select the `dairy_milk.jpg` image and click Open.

The image you selected is displayed in Design view. The source property is also added to the MXML tag.

- 7 Click the Scale Mode drop-down menu and change the value to `letterbox`.

In an ideal world, all the images that you use in the application would be a perfect size, but this is not always the case. Flex can scale the images in two ways. You can choose `letterbox` to keep the aspect ratio of the original images correct even as their size is adjusted, or you can choose `stretch` to distort the images to make them fit into any given width and height.



- 8 Switch back to Source view and notice that Flash Builder has added an `<s:Image>` tag as well as the attributes you specified in the Flex Properties window.

*** NOTE:** Letterbox is the default selection if you don't choose a Scale Mode. So, if you didn't explicitly choose it from the drop-down list and instead left it as the default, you may not see it in your code. Feel free to add it or just understand that difference going forward.

As you can see, it is easy to switch between Source view and Design view, and each one has its advantages. Notice as you switch back to Source view that the Image tag you were working on is now highlighted.

```

25< <s:VGroup id="products" width="100%" height="150"
26     visible.cartView="false" width.cartView="0" height.cartView="0">
27     <s:Label id="prodName" text="Milk"/>
28     <s:Image includeIn="State1" scaleMode="letterbox" source="assets/dairy_milk.jpg"/>
29     <s:Label id="price" text="$1.99"/>
30     <s:Button id="add" label="Add To Cart"/>
31 </s:VGroup>

```

- 9 In the `<s:Image>` tag that you added, insert an `@Embed` directive to the Image control.

```

<s:Image includeIn="State1" scaleMode="letterbox"
  ➤ source="@Embed('assets/dairy_milk.jpg')"/>

```

The `@Embed` directive causes the compiler to transcode and include the JPG in the SWF file at compile time. This technique has a couple advantages over the default of loading the image at runtime. First, the image is loaded at the start of the application, so the user doesn't have to wait for the image to load before displaying when it is needed. Also, this technique can be useful if you are building offline applications that do not need to access the Internet because the appropriate images are included in the SWF file and will be correctly displayed when needed. Remember, though, that using this technique greatly increases the size of your SWF file.

- 10 Save, compile, and run the application.

You should see that the Image and Label controls and button fit neatly into the layout container.



Building a Detail View

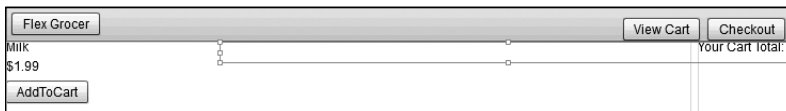
In this exercise, you will use a rollover event to display a detailed state of the application. You will explore different simple controls to display text and review how application states work.

- 1 Be sure that you are still in Source view in Flash Builder. Near the top of the file, locate the `<s:states>` block, which contains definitions for the `State1` and `cartView` states. Add a new state definition named `expanded`.

```
<s:State name="expanded"/>
```

You will define this third state for the application to show details of a product.

- 2 Switch to Design view, set the state selector to **expanded**, and drag a `VGroup` from the Layout folder of the Components view into the application. (To position this correctly, you should drag the `VGroup` into the gray area below the existing white background.) In the Properties view, verify that the `In` state's value is `expanded`, the `X` value is 200, and the `Width` value is 100 percent. Remove the `Y` and `Height` values so that the fields are blank.



This new `VGroup` needs to be a child of the main application. Sometimes, positioning items correctly can be difficult in Design view, so switch to Source view and ensure the `VGroup` is positioned correctly. It should be just above the closing `</s:Application>` tag, so the end of the file reads like this:

```
</s:VGroup>
</s:HGroup>
<s:VGroup includeIn="expanded" width="100%" x="200">
</s:VGroup>
```

```
</s:Application>
```

- 3 Switch back to Design view. Ensure that the `expanded` state is selected in the States view. Drag an instance of the `RichText` control from the Controls folder of the Components view into the new `VGroup` you created in the previous step.



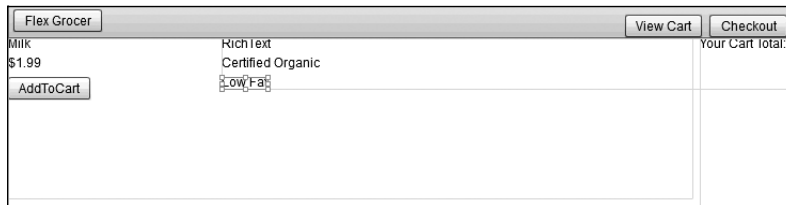
The RichText control enables you to display multiple lines of text, which you will need when you display the product description that will ultimately come from an XML file. You will use data binding in the next section to make this RichText control functional. For now, you are just setting up the layout.

- 4 Drag an instance of the Label control from the Components view to the bottom part of the VGroup container you created. Populate the text property with the words **Certified Organic**.

Later on, you will modify the `visible` property of this component so the contents of the text property are displayed only when a grocery item is certified organic.



- 5 Drag another instance of the Label control from the Components view to the bottom part of the VGroup container you created. Populate the text property with the words **Low Fat**.



Later, you will set the `visible` property of this label to `true` if the grocery item is low fat, or `false` if it is not.

- 6 Switch back to Source view. Notice that Flash Builder has added the RichText and the two Label controls you added in Design view.

Note that all the code created in Design view is displayed in Source view.

- 7 Locate the `<s:RichText>` tag in the expanded state and set the `width` property to 50%.

```
<s:RichText text="RichText" width="50%" />
```

- 8** Find the `<s:Image>` tag that is displaying the milk image. Add a `mouseover` event to the tag that will change the `currentState` to `expanded`. Remove the `includeIn` attribute.

```
<s:Image scaleMode="Letterbox"
        source="@Embed('assets/dairy_milk.jpg')"
        mouseOver="this.currentState='expanded'"/>
```

`mouseover` simply means that when the user rolls the mouse anywhere over the `dairy_milk.jpg` Image tag, the ActionScript will execute. In this ActionScript, you are referring to the `expanded` state, which you created earlier in this lesson.

If you had left the `includeIn` attribute in the image tag, the milk image would appear only in the initial state of `State1`. Therefore, when you mouse over the image and switch it to the `expanded` state, the milk bottle image will disappear. By removing the `includeIn` attribute, you are instructing the application to allow this image to be used in all states.

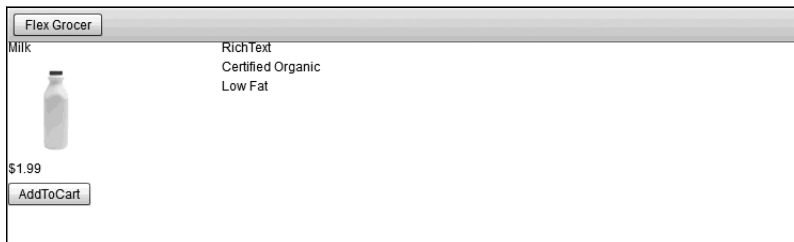
- 9** In the same `<s:Image>` tag, add a `mouseout` event that will change the `currentState` back to the initial `State1` state.

```
<s:Image scaleMode="Letterbox"
        source="@Embed('assets/dairy_milk.jpg')"
        mouseOver="this.currentState='expanded'"
        mouseOut="this.currentState='State1'"/>
```

When the user moves the mouse away from the `dairy_milk.jpg` image, the detailed state no longer displays, and by default the application displays only the images and labels for the control, which is expressed with an empty string.

- 10** Save and run the application.

When you roll the cursor over the milk bottle image, you see the RichText and Label controls you created in the `expanded` state.



Using Data Binding to Link a Data Structure to a Simple Control

Data binding enables you to connect controls, such as the text controls that you have already worked with, to an underlying data structure. Data binding is incredibly powerful because if the underlying data changes, the control reflects the changes. For example, suppose you create a text control that displays the latest sports scores; also suppose it is connected to a data structure in Flex. When a score changes in that data structure, the control that the end user views reflects the change. In this exercise, you will connect a basic data structure in an `<fx:Model>` tag to simple UI controls to display the name, image, and price for each grocery item. Later in the book, you will learn more about data models, the effective use of a model-view-controller architecture on the client, and how to connect these data structures with server-side data.

- 1 Be sure that `FlexGrocer.mxml` is open, and add an `<fx:Model>` tag after the comment in the `<fx:Declarations>` tag pair at the top of the page.

The `<fx:Model>` tag allows you to build a client-side data model. This tag converts an XML data structure into a format that Flex can use.

- 2 Directly below the opening `<fx:Model>` tag and before the closing `<fx:Model>` tag, add the following XML data structure. Your `<fx:Model>` tag should look as shown:

```
<fx:Model>
  <groceries>
    <catName>Dairy</catName>
    <prodName>Milk</prodName>
    <imageName>assets/dairy_milk.jpg</imageName>
    <cost>1.20</cost>
    <listPrice>1.99</listPrice>
    <isOrganic>true</isOrganic>
    <isLowFat>true</isLowFat>
    <description>Direct from California where cows are happiest!</description>
  </groceries>
</fx:Model>
```

You have defined a very simple data structure inline inside an `<fx:Model>` tag.

- 3 Assign the `<fx:Model>` tag an ID of `groceryInventory`. The first line of your `<fx:Model>` tag should look as shown:

```
<fx:Model id="groceryInventory">
```

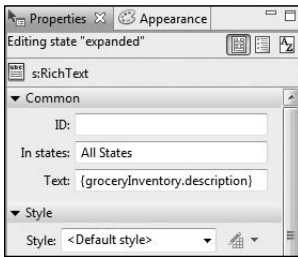
By assigning an ID to the `<fx:Model>` tag, you can reference the data with dot syntax. For example, to access the list price of the item, you could use `groceryInventory.listPrice`. In this case, that would resolve to 1.99.

4 Switch Flash Builder to Design view.

You can set up bindings between elements just as easily in Design view as you can in Source view.

5 Select the RichText control in the expanded state and be sure that the Flex Properties view is open. Modify the text property to `{groceryInventory.description}`.

Data binding is indicated by the curly brackets `{}`. Whenever the curly brackets are used, you use `ActionScript` instead of simple strings. Effective use of data binding will become increasingly important as you begin to work with server-side data.

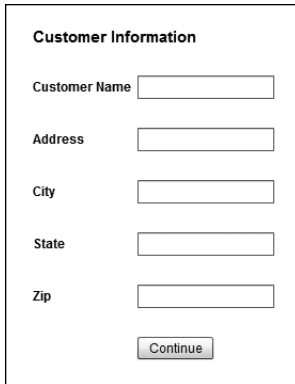


6 Save and run the application.

You should see the description you entered in the data model when you roll the cursor over the grocery item.

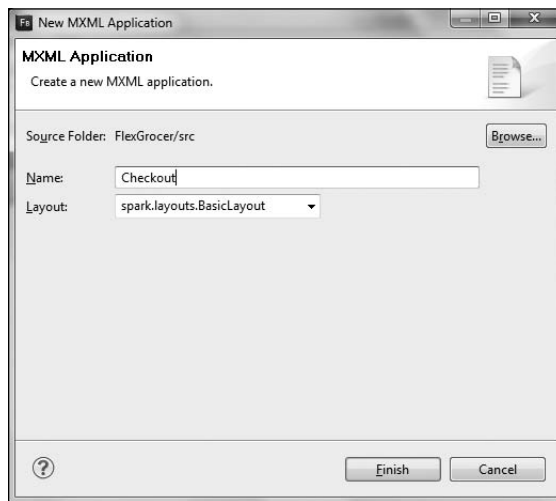
Using a Form Layout Container to Lay Out Simple Controls

Forms are important in most applications that collect information from users. You will be using the Form container to enable shoppers to check out their products from the grocery store. The Form container in Flex will handle the layout of the controls in this form, automating much of the routine work. With a Form container, you can designate fields as required or optional, handle error messages, and perform data checking and validation to be sure the administrator follows designated guidelines. A Form container uses three tags: an `<s:Form>` tag, an `<s:FormHeading>` tag, and an `<s:FormItem>` tag for each item on the form. To start, the checkout form will be built into a separate application, but later in the book, it will be moved into the main application as a custom component.

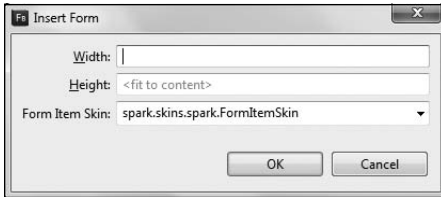


A form titled "Customer Information" with five text input fields and a "Continue" button. The fields are labeled "Customer Name", "Address", "City", "State", and "Zip".

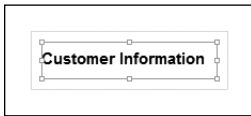
- 1 Create a new MXML application in your current project by choosing File > New > MXML Application. Name the application **Checkout**, and choose spark.layouts.BasicLayout as the layout for the new application. Then click Finish.



- 2 Switch to Design view, and drag a Form from the Layout folder of the Components view to the top left of the window. A dialog box will appear asking for the Width and Height of the form. Leave the default values and click OK.

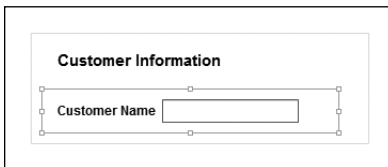


- 3 Drag a FormHeading component from the Layout folder in the Components view into the newly created form. Double-click the FormHeading, and change it to **Customer Information**.



A FormHeading is just a specialized label for Forms.

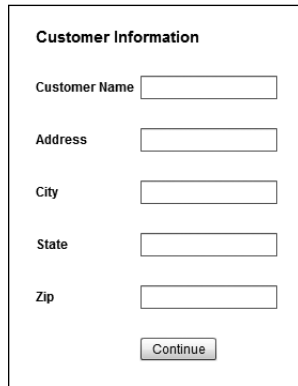
- 4 Drag a TextInput control from the Controls folder of the Components view and drop it just below the FormHeading. The TextInput and a label to the right of the TextInput both appear. Double-click the label and change it to **Customer Name**.



When adding controls to a form in Design view, Flash Builder automatically surrounds the control in a FormItem, which is why a label is appearing to the left of the control. If you switch to Source view, you can see the FormItem surrounding the TextInput. Back in Design view, notice how the left edge of the text input's label is aligned with the left edge of the FormHeading. As noted earlier, this is a feature of the Form and FormHeading classes, and it allows these items to always maintain the left alignment, regardless of the size of the FormItem labels.

- 5 Drag four more TextInputs to the form from the Components view. Change the labels of these to **Address**, **City**, **State**, and **Zip**. Drag a button below the last TextInput, and set its label to be an empty string (simply remove the default text). Click the button and change the button's text to **Continue**.

Due to the Form layout, selecting a discrete control such as the Button can be difficult. In this case, it is easiest if you attempt to click the very left side of the button. Remember, if you can't accomplish the desired effect in Design view, you can always do so in Source view.



The image shows a rectangular form titled "Customer Information". Inside the form, there are five text input fields, each preceded by a label: "Customer Name", "Address", "City", "State", and "Zip". Below these fields is a button labeled "Continue".

Each control is surrounded in its own FormItem and has its own label. Since you don't need a label next to the Continue button, you simply clear the text from the label on that form item.

- 6 Save and run the application.

What You Have Learned

In this lesson, you have:

- Learned how to load images at runtime with the Image control (pages 81–84)
- Learned how to display blocks of text (pages 85–87)
- Learned how to link simple controls to an underlying data structure with data binding (pages 88–89)
- Learned how to build user forms with a minimum of effort using the Form container (pages 89–92)

This page intentionally left blank

LESSON 5

What You Will Learn

In this lesson, you will:

- Learn about Flex's event-based programming model
- Pass data to an event handler using MXML
- Learn a bit about UI object creation order
- Handle both user and system events with ActionScript functions
- Understand the event object and its properties

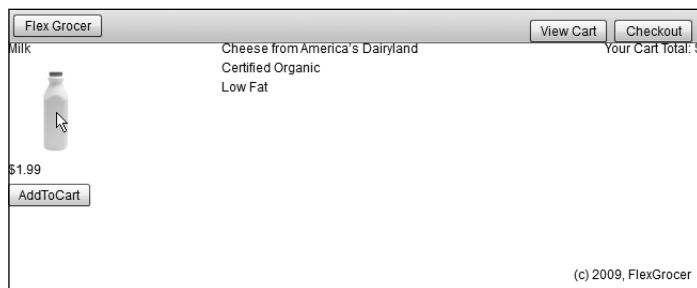
Approximate Time

This lesson takes approximately 1 hour to complete.

LESSON 5

Handling Events

An important part of building a rich Internet application (RIA) is building an effective client-side architecture. When you use Flash Builder as an authoring tool, you have the ability to follow object-oriented best practices and an event-based programming model that allows for loosely coupled applications. This type of development is very different for web application developers, because it does not follow a page-based, flow-driven model. Ultimately, using this client-side, event-based architecture can result in better-performing applications that contain more reusable code and consume less network traffic because page refreshes are no longer needed. During this lesson, you'll become familiar with an event-based programming model and learn how events are used throughout Flex.



Events are added to the FlexGrocer application to allow the user to interact with the application.

Understanding Event Handling

Flex uses an event-based, or event-driven, programming model: Events determine the flow of the application. For example, a user clicking the mouse button or a server returning data determines what should happen next.

These events come in two types: user events and system events. User events are just what you'd most likely guess—a user clicking a mouse or pressing a key. System events include the application being instantiated, an invisible component becoming visible, and many others. The Flex developer handles these events by writing code for what happens next.

▶ **TIP:** Many server-side developers are accustomed to a flow-driven programming model, in which the developer determines the flow of the application rather than having to react to events generated by the user or system; recognizing and understanding that difference is crucial to success with Flex.

For the moment, we are going to personify Flex's event-based model to make its operation clear. Pretend that you and a friend are standing on opposite sides of a parking lot. Your friend is going to act as an *event dispatcher*, an object that notifies others when something occurs.

While you're in the parking lot, your friend may shout a variety of things. He may exclaim, "Car arriving!" "Car departing!" or "Car parking!" Perhaps he periodically decides to simply yell, "Nothing has changed!" In all cases, he is shouting the information, and anyone close can hear it. He has no real control over who hears it, and certainly no control over what a person who overhears these messages might do in response. His only job in this scenario is to announce information, which is precisely the job of an event dispatcher.

Now, on the other side of the parking lot, you hear these various messages being announced. You may choose to react to all, some, or none of them. When you hear that a car is parking, for example, you may wish to go and greet the new arrival. However, you may blatantly ignore your friend announcing that nothing has changed or that a car is departing.

In this case, you are an *event listener*. While you can hear all the information being broadcast, you decide which messages are important and how you react to them. Just as hearing something is different from actually listening to it in the real world, the same difference exists in event-driven programming. Code that listens to and reacts to a given event is called an event listener or an *event handler*.

Now, as a last step in this example, imagine that another individual arrives at the parking lot. He can also hear your friend shouting. He may choose to listen and react to the same mes-

sages as you, or different ones altogether. Perhaps when he hears that a car is departing, he walks up to the car and asks for payment for parking, while ignoring the other messages.

This is the wonderful part about event-based programming. Many people can hear a message, and each can decide whether to react to it. If they do choose to react, they can do so in different ways. And, the person doing the shouting doesn't need to know what might happen as a result; his only job is to keep shouting.

Bringing this back to Flex, we might say that an object such as a `Button` instance dispatches a `click` event. What we mean is that the `Button` shouts for all to hear that it has just been clicked. Every other object in the system can choose to listen to that event and to handle it (react). In Flex, if we don't choose to listen to an event on an object, then we implicitly ignore it.

Keeping that in mind, the following general steps occur when a user event occurs and a developer then wants something to happen:

1. The user interacts with the application.
2. The object on which the user interacts dispatches an event (for example, when a button has been clicked).
3. Another object is listening for that event and reacts when the event occurs.
4. Code inside the listening object is executed.

Analyzing a Simple Example

Let's examine a concrete example: A user clicks a button and text appears in a label. The following code makes this happen.

```
<s:Label id="myL"/>

<s:Button id="myButton"
  label="Click Me"
  click="myL.text='Button Clicked'"/>
```

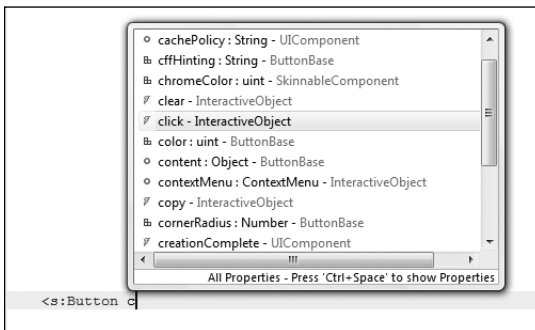
A button appears with a "Click Me" label. When the user clicks the button, the `click` event is dispatched. In this case the ActionScript code `myL.text='Button Clicked'` is executed. The `text` property of the label is assigned the `Button Clicked` string value.

*** NOTE:** There are nested quotes in this example. The double quotes surround the code for the `click` event, and the nested single quotes delineate the string. This can become difficult to read and, as you will see in the next section, is not the ideal way to write code.

Until now, when you've assigned values to an attribute in MXML, you've supplied one of two types: scalar values or bindings. *Scalar values* are simple data types like String, Number, or Boolean values. You've used these when setting *x* and *y* values, widths, and label text values. You have also used bindings for properties. This was done whenever you used braces ({}) in a value. Remember from the last lesson that the braces let you enter ActionScript for the property value.

When supplying a value to an MXML attribute that represents an event (that is, the `click` event in the previous code), the Flex compiler is smart enough to implicitly understand that the string inside the quotes is ActionScript. That's why you can enter ActionScript directly for the `click` event, `click="myL.text='Button Clicked'"`, without using the braces you used in the previous lesson.

Just as code hinting assisted you when you were entering property names, so code hinting will assist with event names. In the following figure, you see the `clear` and `click` events displayed with the lightning bolt icon in front of them, which designates events.



Handling the Event with an ActionScript Function

The code in the last example successfully sets the text of the Label object when the Button is clicked. However, a problem with this approach soon develops when you want to execute more than one line of ActionScript when the event occurs. To do this, you could place many separate chunks of code inside the quotes for the `click` event, each separated by a semicolon. Although this works, it's messy and far from a best practice. Also, you may want the same lines of code to be executed when several different events occur. In the approach shown earlier, you would have to copy and paste the same code into several places. That can become a nightmare if you ever need to edit that code, as you now need to find and edit each copy.

A better approach is to handle the event in an ActionScript function. The function will be built in an `<fx:Script>` block that simply tells the Flex compiler that the code in the Script block is ActionScript. So instead of placing the actual ActionScript to be executed as a value for the `click` event, you will call a function. Following is a refactoring of the earlier code, using the best practice of placing the code to be executed in a function.

```
<fx:Script>
  <![CDATA[
    private function clickHandler():void {
      myL.text="Button Clicked";
    }
  ]]>
</fx:Script>

<s:Label id="myL"/>

<s:Button id="myButton"
  label="Click Me"
  click="clickHandler()"/>
```

*** NOTE:** The `<![CDATA[]]>` block inside the Script block marks the section as character data. This tells the compiler that the data in the block is character data, not well-formed XML, and that it should not show XML errors for this block.

Now when the Button is clicked, the function `clickHandler()` is called, and the string is written to the label. In this case, because no quotes were nested, you can use double quotes around the string in the function.

The function has a return type of `void`. This means that the function will not return a value. It is a best practice to always specify the return type of functions you build, even if you use `void` to indicate that no data will be returned. The compiler will give you a warning if you do not specify a return type on a function. It is best to heed those warnings, as specifying types enables the compiler to ensure that you don't make simple typos, like assigning a variable that is supposed to contain a Button to something that is supposed to contain a Number.

Passing Data When Calling the Event Handler Function

You may wish to pass data when calling the function. This works in ActionScript just as you'd expect. You place the data to pass inside the parentheses following the function name, and then modify the event handler to accept the parameter. Just as you will always specify a return type on your function, so will you need to specify the type for any parameter that the function will accept.

In the following code example, the string to be displayed in the label is passed to the `clickHandler()` when the button is clicked.

```
<fx:Script>
  <![CDATA[
    private function clickHandler(toDisplay:String):void {
      myL.text=toDisplay;
    }
  ]]>
</fx:Script>

<s:Label id="myL"/>

<s:Button id="myButton"
  label="Click Me"
  click="clickHandler('Value Passed')"/>
```

In this case, when the Button is clicked, the string `Value Passed` is sent to the event handler function. The function accepts the parameter in the `toDisplay` variable, which has a data type of `String`. The value stored in the `toDisplay` variable is then displayed in the label's text property.

Using Data from the Event Object

So far you've examined a few different ways of handling events but, before you try it yourself, there is one last item to understand, the *event object*. When personifying the event model, we discussed it in terms of a message being shouted. In reality, when an event is dispatched, it is more than just a message; it's an entire object. This object, referred to as the event object, can have many different properties.

The most basic event in the Flex world is the aptly named `Event` class. This is an `ActionScript` class that defines only the most basic properties needed to be an event. The most important among these properties are `type`, which is a `String` containing the name of the event (the message being shouted)—for example, `click` or `creationComplete`—and the `target`, which is the component dispatching the event (your friend shouting).

*** NOTE:** *Target* may seem like an odd name for this property. It might be more aptly named *source*, as it refers to the object that broadcasts the event. This property name will make a little more sense once you finish Lesson 11, "Creating Event Classes," and learn about event flow.

In practice, subclasses of `Event` are used much more often than the `Event` class. Imagine a situation in which you drag an item from one place on the screen to another. Knowing that an item was dragged, and to where it was dragged, are both important, but you would likely want

some additional information as well: what item was being dragged, for example, and what the x and y positions of the mouse were when the item was dropped. To provide this more specific information, Event subclasses and additional properties are added, meaning you will often interact with event types such as DragEvents or ResultEvents. The following figure from the documentation shows how many other event classes are based on, or subclassed from, the generic Event object.

Package	flash.events
Class	public class Event
Inheritance	Event → Object
Subclasses	ActivityEvent, AdvancedDataGridEvent, AdvancedDataGridHeaderShiftEvent, AdvancedDataGridItemSelectEvent, AIREvent, AutomationEvent, AutomationRecordEvent, AutomationReplayEvent, BrowserChangeEvent, BrowserInvokeEvent, CalendarLayoutChangeEvent, CaptionChangeEvent, CaptionTargetEvent, ChannelEvent, ChartSelectionChangeEvent, ChildExistenceChangeEvent, CloseEvent, CollectionEvent, ColorPickerEvent, ColorPickerEvent, ComponentEvent, CompositionCompleteEvent, ConflictEvent, ContextMenuEvent, CubeEvent, CuePointEvent, DamageEvent, DataChangeEvent, DataConflictEvent, DataGridEvent, DataChooserEvent, DividerEvent, DRMAuthenticateEvent, DRMAuthenticationCompleteEvent, DRMStatusEvent, DropDownEvent, DropDownEvent, DynamicEvent, EffectEvent, ElementExistenceEvent, FileEvent, FileListEvent, FlexEvent, FlexChangeEvent, FlexNativeMenuEvent, FlowElementMouseEvent, FlowOperationEvent, FocusEvent, HTMLUncaughtScriptExceptionEvent, HTTPStatusEvent, IEvent, IndexChangeEvent, IndexChangeEvent, InterManagerRequest, InvokeEvent, ItemClickEvent, KeyboardEvent, LayoutEvent, ListEvent, ListEvent, ListItemSelectEvent, LogEvent, MarshalledAutomationEvent, MenuShowEvent, MessageEvent, MessageFaultEvent, MetadataEvent, MetadataEvent, MotionEvent, MouseEvent, MoveEvent, NativeWindowBoundsEvent, NativeWindowDisplayStateEvent, NetStatusEvent, NumericStepperEvent, OutputProgressEvent, ProgressEvent, PropertyChangeEvent, RenderDataEvent, Request, ResizeEvent, SampleDataEvent, SandboxMouseEvent, ScrollEvent, ScrollEvent, SelectionEvent, SessionFaultEvent, SessionResultEvent, ShaderEvent, SliderEvent, SliderEvent, SoundEvent, SQLUpdateEvent, StateChangeEvent, StateChangeEvent, StatusEvent, SWFBridgeEvent, SWFBridgeRequest, SyncEvent, TextEvent, TextLayoutEvent, TextOperationEvent, TextSelectionEvent, TimerEvent, ToolTipEvent, TrackBaseEvent, TreeEvent, TweenEvent, TweenEvent, UpdateCompleteEvent, UpdateEvent, ValidationResultEvent, VideoEvent, VideoEvent, VideoEvent, XMLLoadEvent

Examine the following code that sends an event object, in this case a MouseEvent object, to the event handler.

```
<fx:Script>
  <![CDATA[
    private function clickHandler(event:MouseEvent):void {
      trace(event.type);
    }
  ]]>
</fx:Script>

<s:Label id="myL"/>

<s:Button id="myButton"
  label="Click Me"
  click="clickHandler(event)"/>
```

In the code, an event is passed to the event handler, and the word *click* will be displayed in the Console view when the application is debugged. You are now going to refactor the FlexGrocer application to use a function for the View Cart buttons.

1 Open the FlexGrocer.mxml file that you created in the previous lesson.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson05/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Directly below the closing `</fx:Declarations>` tag, insert a new `<fx:Script>` tag pair. When you do so, Flash Builder will automatically insert a CDATA (character data) block for you. Your code should look like the following:

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

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

MXML files are XML files. Some of the characters you are about to use when writing ActionScript code are not allowed inside XML directly. The CDATA block instructs XML parsers to treat the data inside it differently, allowing these characters. Throughout this book you will be asked to add functions and variables inside the Script tag. You should always add these inside the CDATA block.

- 3 Inside the Script block (remember that also means inside the CDATA block), add a new private function named `handleViewCartClick()`. This function will accept a single argument named `event` of type `MouseEvent` and return nothing (have a return type of `void`).


```
private function handleViewCartClick( event:MouseEvent ):void {
}
```

As you can see, the first word in this function declaration is `private`. This is the function's scope. Here you are indicating that a function is accessible only from inside this object. You are specifying that a single argument will be passed to this function and that argument will be of type `MouseEvent`.

- * **NOTE:** You will deal more with the scope of functions later in this book. However, there isn't enough room in this book to cover both object-oriented programming and Flex, so if you are uncomfortable with this or any other object-oriented concepts, please review any number of excellent books on that topic, or read the extensive entries on Wikipedia.

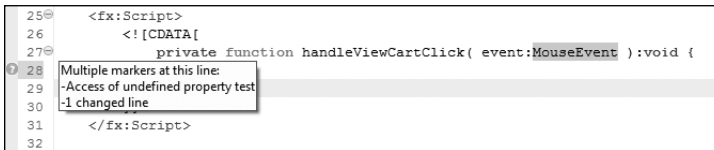
- 4 Inside the `handleViewCartClick()` function, add the word `test`.

```
private function handleViewCartClick( event:MouseEvent ):void {
  test
}
```

If you pause for just a moment, you will see an orange circle with a question mark  appear on the left border of your editing window near the line numbers. This is an indication that Flash Builder does not understand the line of code you just entered and that it may be in error.

Flash Builder provides this feedback continually during development to save you time. Depending on how quickly you type, this orange circle may appear while you are entering a line of code; however, it should disappear when the line is complete.

If you hover over this circle with your mouse, Flash Builder will provide you more information that can be very useful while you are learning Flex.



```

25 </fx:Script>
26 <![CDATA[
27     private function handleViewCartClick( event:MouseEvent ):void {
28     Multiple markers at this line:
29     -Access of undefined property test
30     -1 changed line
31     }
32 </fx:Script>

```

- 5 Delete the word `test` from inside the `handleViewCartClick()` function and add ActionScript code to change the `currentState` property to `cartView`.

```

private function handleViewCartClick( event:MouseEvent ):void {
    this.currentState="cartView";
}

```

- 6 Find the Button with the id of `btnCartView` inside the `controlBarContent`. Currently that Button sets the `currentState` directly. Instead, change this tag so it now calls the `handleViewCartClick()` function and passes the event object.

```

<s:Button id="btnCartView" label="View Cart" right="90" y="10"
    click.State1="handleViewCartClick( event )"/>

```

- 7 Find the Button inside the `cartGroup` with the label `View Cart` that currently sets the `currentState` to `cartView` directly. Change this tag so it now calls the `handleViewCartClick()` function and passes the event object.

```

<s:Button label="View Cart" click="handleViewCartClick( event )"
    includeIn="State1"/>

```

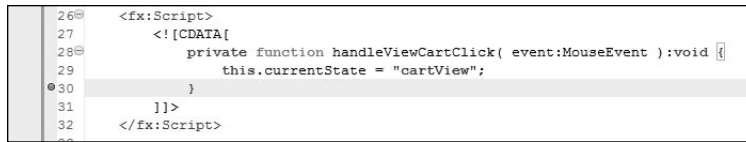
- 8 Save the file and click Run.

As with all refactoring, the application should behave the same as it did previously with both `View Cart` buttons taking the application to the `cartView` state.

Inspecting the Event Object

In this section, you will use the debugger to examine MouseEvent properties. Learning to use the event object and its properties is one key to writing reusable code in Flex.

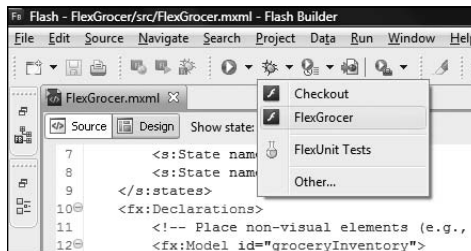
- 1 Add a breakpoint on the closing parenthesis of the `handleViewCartClick()` function by double-clicking in the marker bar just to the left of the code and line numbers. A small blue dot will appear in the marker bar indicating where the program execution will halt. You will be able to examine values at this point.



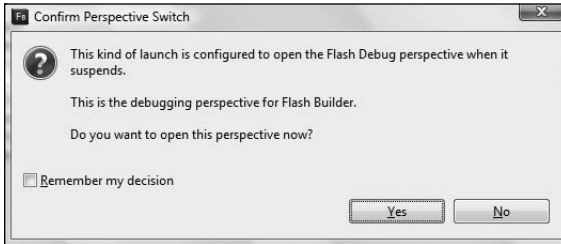
```
26 <fx:Script>
27 <![CDATA[
28     private function handleViewCartClick( event:MouseEvent ):void {
29         this.currentState = "cartView";
30     }
31 ]]>
32 </fx:Script>
```

The debugger is immensely helpful in understanding what is happening in Flex. Use it often to get a sense of what's going on “under the hood” of the application.

- ▶ **TIP:** Event handlers should be named consistently. For instance, in this lesson you've seen a `click` event on two View Cart buttons handled by an event handler named `handleViewCartClick()`. There is no “right” way to name event handlers, but you may wish to pick a naming convention and stick with it. The most important point is to make them as descriptive as possible.
- 2 In the Flash Builder interface, click the down arrow next to the Debug button and choose FlexGrocer.
- ▶ **TIP:** If you have multiple application files, such as FlexGrocer and Checkout, you can choose the specific one you wish to run by clicking the down arrow next to the Run or Debug buttons instead of the button itself.



- 3 In the browser, click either of the buttons labeled View Cart. In Flash Builder, you may be prompted to use the Debug perspective. This dialog box will only appear if you did not select the *Remember Your Decision* check box earlier. If prompted, select the Debug perspective now.

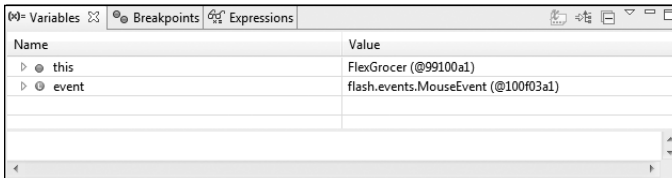


Clicking Yes will switch your Flash Builder view to the Debug perspective, which is optimized with the views needed to debug an application.

- 4 Double-click the tab of the Variables view. This will cause the Variables view to maximize and take up the full screen.

The Variables view can provide an immense amount of information. A full-screen view will make navigating that information easier.

You will see two variables displayed, `this` and `event`, as shown here:

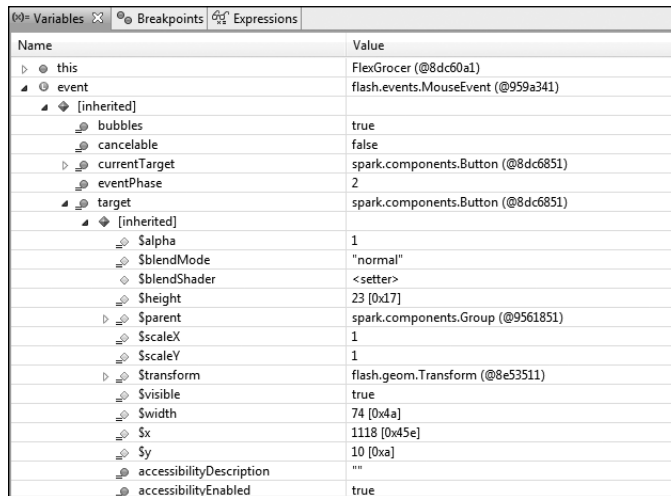


Right now, the `this` variable represents the entire application. When you learn about creating your own components in future lessons, you will learn that `this` always represents a context that can change. If you click the arrow in front of the variable, you will see many properties and associated values. The `event` variable represents the event variable local to the function where the breakpoint was placed. The letter *L* in a circle in front of `event` indicates it is a local variable.

- 5 Click the outline of the arrow to the left of the event variable and then the arrow to the left of the [inherited] set of properties. Locate the target property. Notice that you clicked a button to get here and that the target of this event is the Button component that broadcasts the event. Also notice the `type` property has the value `click`.

From the earlier discussion, neither of those property values should be a surprise. In this case, the values listed in the `[inherited]` section are those available to every event because they come from the superclass. The properties listed outside that section are the specific properties available on the `MouseEvent`. Note that some of these properties, such as `altKey` or `localX`, wouldn't make sense to have on every event in the system, but they are welcome properties on a `MouseEvent`.

- Click the arrow to the left of the target, then click the arrow to the left of the `[inherited]` set of properties. Locate the `id` property.



Name	Value
▶ this	FlexGrocer (@@8dc60a1)
▲ event	flash.events.MouseEvent (@@959a341)
▶ [inherited]	
bubbles	true
cancelable	false
currentTarget	spark.components.Button (@@8dc6851)
eventPhase	2
target	spark.components.Button (@@8dc6851)
▶ [inherited]	
\$alpha	1
\$blendMode	"normal"
\$blendShader	<setter>
\$height	23 [0x17]
\$parent	spark.components.Group (@@9561851)
\$scaleX	1
\$scaleY	1
\$transform	flash.geom.Transform (@@8e53511)
\$visible	true
\$width	74 [0x4a]
\$x	1118 [0x45e]
\$y	10 [0xa]
accessibilityDescription	""
accessibilityEnabled	true

This property will depend on which View Cart button you clicked. If you clicked the button in the control bar, then the property's value is `btnCartView`, which is what you assigned in the code. If you chose the View Cart button in the shopping cart area, then you will see an assigned id such as `_FlexGrocer_Button5`. If you wish, repeat the steps to view the id values for the other Button. All UI objects in Flex have an id. If you don't assign one, then Flex will.

- Double-click the Variables tab again to restore it. Click the red box on either the Debug or Console view to terminate the debugging session.

Don't forget to terminate debugging sessions. It is possible to have one debugging session running alongside another in certain browsers. You might want to do this in special cases, but not normally.

- Return to the Development perspective by clicking the chevron (`>>`) in the top right of your screen and then choosing Flash.

- ▶ **TIP:** If you place the cursor to the left of the Open Perspective icon, the sideways arrow will appear. You can drag to the left to increase the space allotted for perspectives. You will be able to see both the Debug and Flash perspectives and will be able to click their tabs to switch between them.
- 9 Remove the breakpoint on the closing parenthesis of the `handleViewCartClick()` function by double-clicking the blue dot in the marker bar just to the left of the code and line numbers.

Handling System Events

As mentioned earlier, there are two types of events you handle in Flex. First, there are user events, like the `MouseEvent` that you handled in the previous section. Second, there are system events that are dispatched by the Flex framework in response to a change in an internal condition. In this section, you will see one of the most common system events related to startup and understand a bit about its use.

Understanding the `creationComplete` Event

The `creationComplete` event is one of many useful events dispatched by Flex components. This event is dispatched when a component has been instantiated and knows both its size and position in the application. The `creationComplete` event of a parent component is always dispatched after all its children have dispatched their `creationComplete` events. In other words, if you have a `Group` with several `Buttons` inside it, each of the `Buttons` will dispatch its `creationComplete` event before the `Group` dispatches its own `creationComplete` event.

Let's examine the following code snippet:

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

  <fx:Declarations>
  </fx:Declarations>

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
```

```
<fx:Script>
  <![CDATA[
    private function addToTextArea(eventText:String):void {
      var existingText:String = reportEvents.text;
      reportEvents.text = existingText + eventText + "\n";
    }
  ]]>
</fx:Script>

<s:TextArea id="reportEvents" width="200" height="100"/>

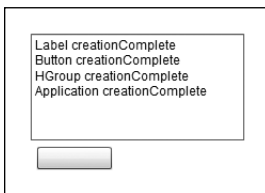
<s:HGroup creationComplete="addToTextArea('HGroup creationComplete')">
  <s:Label creationComplete="addToTextArea('Label creationComplete')"/>
  <s:Button
creationComplete="addToTextArea('Button creationComplete')"/>
</s:HGroup>
</s:Application>
```

First, look at the event handler named `addToTextArea`. This event handler simply accepts a parameter named `eventText` and places it in a `TextArea`, followed by a return so the text doesn't all run together. In each component, which includes `Application`, `HGroup`, `Label`, and `Button`, a `creationComplete` event is handled. When each component finishes its completion process, the event is dispatched and the corresponding string is sent to the event handler for display in the `TextArea`.

Flex begins creating components from the outside, working its way in. However, `creationComplete` means a component has been instantiated and knows both its size and position. In Flex, the size of a component (such as the `HGroup`) is often dictated by the collective size of its children. Therefore, the `Label` and `Button` must be finished (and must have dispatched their `creationComplete` events) before the `HGroup` can be considered complete and can dispatch its own event.

When all the `Application`'s children are created, the `Application` dispatches its `creationComplete` event.

The results displayed in the `TextArea` appear as shown here:



Armed with this knowledge, you can probably understand why the `creationComplete` event of the `Application` object is often used for doing work such as modifying or retrieving data.

Modifying Data on Creation Complete

Currently your FlexGrocer project uses data binding to populate a RichText control with data from an XML model. Your code looks like this:

```
...
<fx:Model id="groceryInventory">
  <groceries>
    <catName>Dairy</catName>
    <prodName>Milk</prodName>
    <imageName>assets/dairy_milk.jpg</imageName>
    <cost>1.20</cost>
    <listPrice>1.99</listPrice>
    <isOrganic>true</isOrganic>
    <isLowFat>true</isLowFat>
    <description>Direct from California where cows are happiest!</description>
  </groceries>
</fx:Model>
...
<s:RichText text="{groceryInventory.description}" width="50%"/>
...
```

Flex knows to automatically populate the RichText control with the data retrieved from the description property of the groceryInventory object. In this section, you will use the creationComplete event to make a small modification to the description in ActionScript and see that the RichText control displays the modified data.

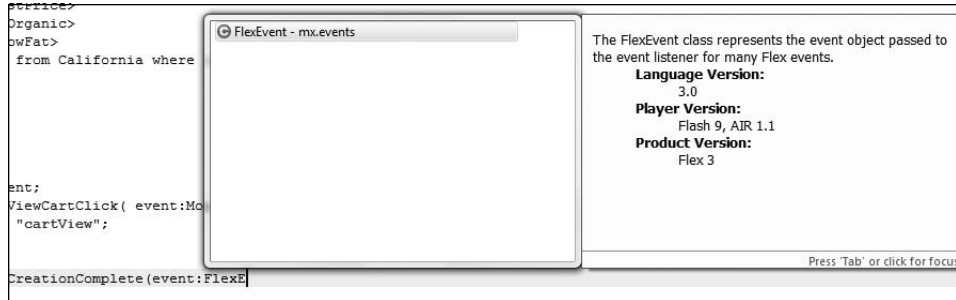
- 1 Open the FlexGrocer.mxml file that you modified in the previous section.

If you didn't complete the previous section, you can import FlexGrocer-PreCreationComplete.fxp project from the Lesson05/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Inside the Script block, just below the existing handleViewCartClick() function, add a new private function named handleCreationComplete(). The function will accept a single argument, named event of type FlexEvent, and return void.

```
private function handleCreationComplete(event:FlexEvent):void {
}
```

While you are typing FlexEvent, Flash Builder will try to provide possible choices as you type. If you choose one of the items on the pop-up menu (or use the arrow keys and press Enter on the correct option), Flash Builder will complete the name for you and perform one other very important step, importing the class.



- 3** If you choose one of the options on the pop-up menu, Flash Builder adds an import line to the top of your Script block. This line is an `import` statement that lets Flash Builder know where the class you are referencing resides. You can think of `import` statements as more or less the ActionScript equivalent of the namespaces you used in MXML:

```
import mx.events.FlexEvent;
```

If you do not have this line in your file, you have two options: You can place your cursor right after the closing `t` in `FlexEvent` and press `Ctrl+Spacebar`. This will cause Flash Builder to open the code-completion pop-up again. If there is only one matching option, Flash Builder automatically selects it and adds the import for you. Alternatively, you can just type the `import` statement just inside the Script tag (remember that also means inside the CDATA block).

- 4** Inside this function, you will assign the string “Cheese from America’s Dairyland” to the `description` property of the `groceryInventory` object.

```
private function handleCreationComplete(event:FlexEvent):void {
    groceryInventory.description = "Cheese from America's Dairyland";
}
```

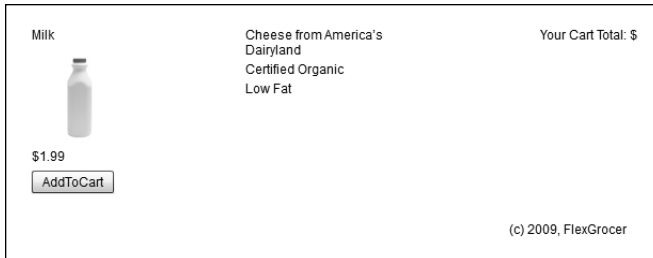
This statement will replace the original text of the `description` property with your new text as soon as the Application dispatches its `creationComplete` event.

- 5** Inside the Application tag, you will instruct Flex to call the `handleCreationComplete()` function when the `creationComplete` event occurs, passing it the event object. Your code should read like this:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="handleCreationComplete(event)">
```

6 Save and run the application.

When you move your mouse over the bottle of milk, you should see the new description appear.



In this simple example, you have handled the `creationComplete` event to modify data and allowed data binding to provide the changed data to the RichText control for display.

What You Have Learned

In this lesson, you have:

- Gained an understanding of event handling in Flex (pages 96–99)
- Learned to pass arguments to an event handler (pages 99–100)
- Refactored the application to use an ActionScript event handler (pages 101–103)
- Handled a `creationComplete` event (pages 109–111)

LESSON 6

What You Will Learn

In this lesson, you will:

- Externalize your data
- Distinguish embedded and loaded data
- Create an HTTPService object that returns data as Objects
- Understand security issues involved with retrieving data into Flash Player
- Search XML with E4X expressions
- Create an HTTPService object that returns data as XML
- Build an XMLListCollection from your dynamic XML
- Display your data in a List

Approximate Time

This lesson takes approximately 1 hour and 30 minutes to complete.

LESSON 6

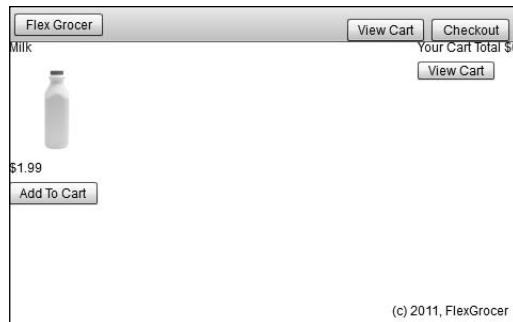
Using Remote XML Data

In this lesson, you will begin to connect the FlexGrocer application to XML data.

First, you will use a local XML file that you will embed in your application. This will demonstrate one technique used to move your XML data into a separate file. Then you will use the `HTTPService` class to load remote XML data into the Application. In this context, the word *remote* means that the data is remote to the application: in other words, not embedded. The data can exist on a remote server or in external files on the same server, but in either case the data is transmitted through HTTP.

You will work with this XML data in several formats, including functionality from the ECMAScript for XML (E4X) implementation that allows you to use XML as a native data type in ActionScript (it's built into Flash Player just like Number, Date, or String).

Controls can be populated with this data to easily display complex datasets and enable the user to navigate the data quickly. Examples of these controls include List, ComboBox, and Tree. You will be using the List control in this lesson.



You will use XML Data to populate the application with live data.

Using Embedded XML

In this task, you will make two major changes: externalizing your data (defining it somewhere external to your Application class) and treating it as XML at runtime.

Currently the XML for your Milk product is hard-coded directly into the Application class in an `<fx:Model>` tag. Hard-coding XML is extremely convenient when you're prototyping new code, but it clutters up your application and can make it difficult to see the important code as new and different types of data are needed.

Externalizing the Model

Your first task is to externalize the data in your application and reference it by filename.

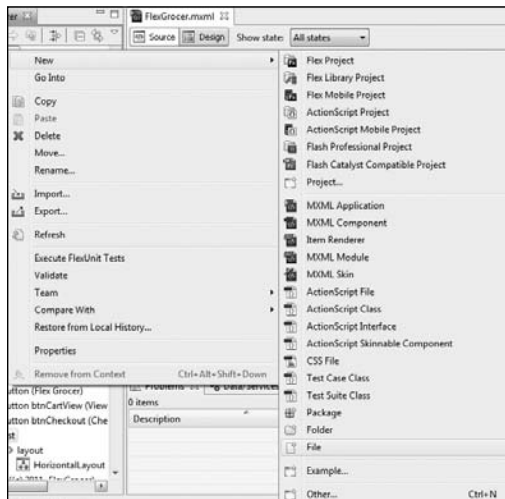
1 Open the FlexGrocer.mxml file.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson06/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

2 Make sure Flash Builder is in Source view.

To switch between Design view and Source view in Flash Builder, click the buttons in the menu bar near the top of the window.

3 Expand the FlexGrocer project and the src folder inside the Package Explorer. Right-click the assets folder and choose New > File.



*** NOTE:** If you are using a Mac, please excuse our PC-centric use of the term right-click and use Control-click instead.

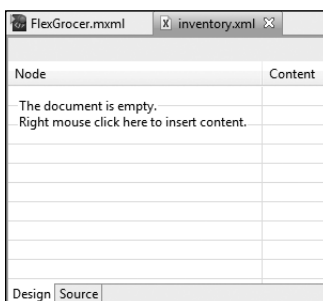
4 Enter **inventory.xml** in the File name field. Click Finish.

You will move the XML specified in your Application file to the external file you just created (inventory.xml) to declutter your Application code.



5 Click the Source tab on the bottom of the inventory.xml file.

When the new inventory file opens in Flash Builder, it will open by default in a type of design view for XML. Going to Source view here will allow you to interact with the XML directly as text.



- 6 On the first line of the new file, you need to add an XML document type declaration that specifies that this file is a version 1.0 XML document encoded with utf-8.

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

This is the same declaration used at the top of each of your MXML files. If you don't want to type it, you can copy it from the first line of the FlexGrocer.mxml file.

- 7 Copy the groceries XML node from within the Model tag of your application. This is the XML that starts with `<groceries>` and ends with `</groceries>`, including both of those tags.
- 8 Paste that XML node into the inventory.xml file directly below the document type declaration. Your inventory.xml file should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<groceries>
  <catName>Dairy</catName>
  <prodName>Milk</prodName>
  <imageName>assets/dairy_milk.jpg</imageName>
  <cost>1.20</cost>
  <listPrice>1.99</listPrice>
  <isOrganic>true</isOrganic>
  <isLowFat>true</isLowFat>
  <description>Direct from California where cows are happiest!</description>
</groceries>
```

- 9 Save the inventory.xml file. You will now use this external file in place of the hard-coded XML in the application.
- 10 Switch back to your FlexGrocer.mxml file and delete the groceries XML from inside the Model. The tag should now be empty.

```
<fx:Model id="groceryInventory">
</fx:Model>
```

- 11 Inside the Model tag, add a source attribute and set it to `assets/inventory.xml`.

```
<fx:Model id="groceryInventory" source="assets/inventory.xml">
</fx:Model>
```

Specifying the source here tells the Model tag to use your external file as its model.

- 12 Finally, change the Model tag to be self-closing. You now want the content in the source file, not content inside the Model tag, to be used for the inventory data.

```
<fx:Model id="groceryInventory" source="assets/inventory.xml"/>
```

- 13 Save and run your application.

The product description and information should work just as they did before; however, the data is now coming from the inventory.xml file.

Choosing Between Objects and XML

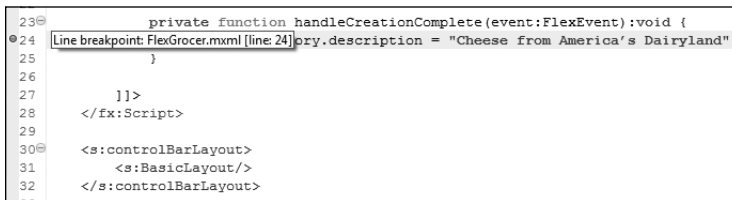
When working in Flex with XML data, you have two choices: Work directly with the XML or convert the XML to an object, then use that object instead of the XML.

The Model tag you've been working with so far does the latter. It is using the XML you typed into the inventory.xml file as a template, from which it creates a series of generic objects. In this exercise you'll see this structure in the debugger and then change the code to use XML directly.

- 1 Open the FlexGrocer.mxml file.

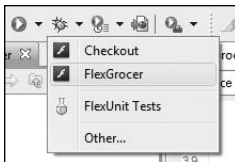
Alternatively, if you didn't complete the previous exercise or your code is not functioning properly, you can import the FlexGrocer-PreXMLTag.fxp project from the Lesson06/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Add a breakpoint inside the `handleCreationComplete()` method by double-clicking in the marker bar just to the left of the code and line numbers. A small blue dot will appear in the marker bar indicating where the program execution will halt.



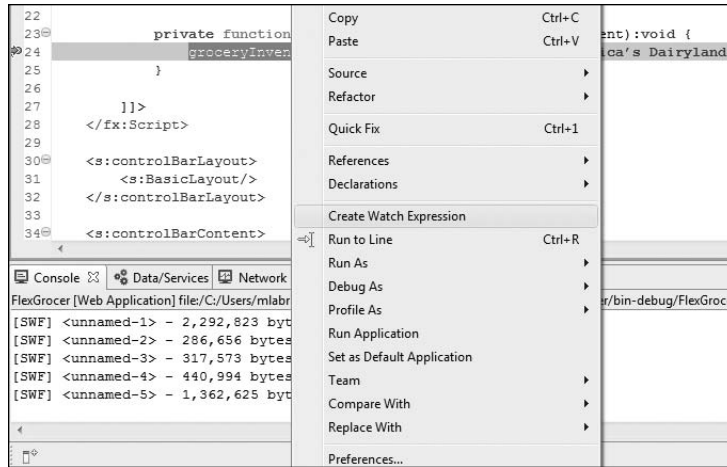
```
23 private function handleCreationComplete(event:FlexEvent):void {
24     [Line breakpoint: FlexGrocer.mxml [line: 24]]    pry.description = "Cheese from America's Dairyland";
25 }
26
27     ]]>
28 </fx:Script>
29
30 <s:controlBarLayout>
31     <s:BasicLayout/>
32 </s:controlBarLayout>
```

- 3 Debug the FlexGrocer application.



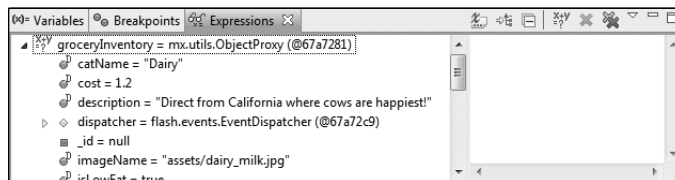
In Flash Builder, you may be prompted to use the Debug perspective. This dialog box will only appear if you did not select the Remember Your Decision check box earlier. If prompted, select the Debug perspective now. Click Yes. Flash Builder will stop on the line where you set a breakpoint.

- 4 Select `groceryInventory` and then right-click it. Choose Create Watch Expression.



Flash Builder will add `groceryInventory` to the Expressions view. If you can't find the Expressions view, go to `Window > Expressions`.

- Expand the `groceryInventory` object by clicking the triangle to the left of it in the Expressions view. Note that the item type is `mx.util.ObjectProxy`.



`ObjectProxy` instances are a special type of wrapper for Objects. They effectively enable Objects to dispatch events, which is important for data binding, as you will learn in future lessons.

The `Model` tag converted your XML to Objects. This is actually its intended purpose, to provide the developer with a quick way to define potentially hierarchical objects using XML. As mentioned previously, this is one of two ways that you can choose to deal with XML in Flex. The other is to leave it as XML and manipulate it directly. You will do that next.

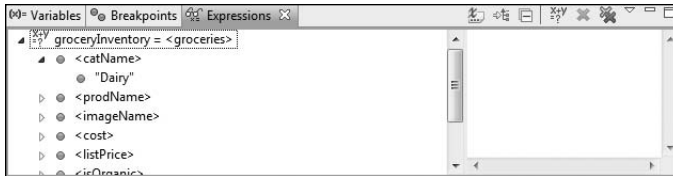
- Click the red square to terminate this debugging session.
- Change the `<fx:Model/>` tag to an `<fx:XML/>` tag.


```
<fx:XML id="groceryInventory" source="assets/inventory.xml"/>
```

- 8** Debug the FlexGrocer application again.

Flash Builder will stop on the line where you set a breakpoint.

- 9** The watch expression you set for `groceryInventory` is still active. So, if you expand the view of this object in the Expressions view, you will see that it is no longer an `ObjectProxy`, but now XML.



- 10** Click the Resume button (the green Play icon to the left of the red Terminate button), and your application will continue to run in the web browser.



- 11** In your web browser, move your mouse pointer over the milk image, and you will see that the description appears as it did before.

The description still displays thanks to a feature of Flash Player called ECMAScript for XML (E4X). It allows you to access data inside XML in much the same way you access other objects, greatly simplifying the task of using XML and making your existing code able to use either objects or XML.

- 12** Terminate your debugging session by clicking the red Terminate button. Remove your breakpoint before continuing.

Your application is now using XML directly instead of Objects for this data. The decision to use one or the other inside Flex is a recurring theme, as you will see in the following exercises.

Using XML Loaded at Runtime

In the previous exercise, you used an external XML file both as an Object and as XML to provide data for your application. However, in both cases, the data was embedded in the application. In other words, the data in your external `inventory.xml` file became part of the final SWF file produced by Flash Builder.

If the data in your `inventory.xml` file changes, you will need Flash Builder to recompile your FlexGrocer application before the changes become available to someone running the

application. This is fine for demo software and potentially even certain pieces of data that do not change very often (for example, the states in the United States). However, it is not practical for most cases.

In this section you'll learn to load data from an external source using the `HTTPService` so that your data can change independent of your application.

Simply stated, the `HTTPService` component allows your application to use data it retrieves at a given URL. By default, the data will be returned in an Object format (as it was when you embedded your file with the `Model` tag). You can choose the format in which you wish to use the data (for example, Object, XML, or text). In this section you'll use the returned data in both the Object and XML formats.


The general steps for using `HTTPService` follow:

1. Create an `HTTPService` object.
2. Invoke the `send()` method of the object.
3. Use the returned data.

Creating an HTTPService Object

You create the `HTTPService` object in the same way as other objects in MXML. When you create the object, you need to specify the URL that the service should access, and potentially specify a method that should be called when a result is retrieved. This is accomplished using the `result` event, which is broadcast when data is successfully returned by the `HTTPService` object. An example of using the `HTTPService` object is shown here:

```
<s:HTTPService id="unitData"
  url="http://www.flexgrocer.com/units.xml"
  result="resultHandler(event) "/>
```

 **TIP:** The `url` property can be an HTTP URL, or even a file URL that points to a file on the file system.

Invoking the send() Method

When you wish to retrieve data from a given URL, you must send a request for that data. The `HTTPService` object contains a method to send this request, named `send()`. When you create the `HTTPService` and specify the URL, the `HTTPService` is ready to retrieve your data; however, it will not begin this process until you invoke the `send()` method. In many cases, you will want to retrieve data at application startup. In this lesson, you'll use the `creationComplete` event of the `Application` tag to retrieve remote data.

Accessing the Returned Data

Data retrieved from the HTTPService can be accessed in two ways.

lastResult

The first is to access the data directly via the `lastResult` property of the named HTTPService object. To get to the data, build an expression with the following elements:

- The instance name of the HTTPService
- The `lastResult` property
- The dot path into the data you are trying to access

For example, in the next exercise you have an HTTPService defined as

```
<s:HTTPService id="unitRPC"
  url="http://www.flexgrocer.com/units.xml"/>
```

and it will retrieve the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<allUnits>
  <unit>
    <unitName>Bunch</unitName>
    <unitID>4</unitID>
  </unit>
  <unit>
    <unitName>Dozen</unitName>
    <unitID>2</unitID>
  </unit>
  <unit>
    <unitName>Each</unitName>
    <unitID>1</unitID>
  </unit>
  <unit>
    <unitName>Pound</unitName>
    <unitID>3</unitID>
  </unit>
</allUnits>
```

By default, it will turn that XML into a series of objects. So, to access the unit node data via the `lastResult` property, you would use the following code:

```
unitRPC.lastResult.allUnits.unit
```

This is the instance name of the HTTPService (`unitRPC`), followed by the `lastResult` property, followed by the path to the piece of data you care about (`unit`).

This method of accessing the returned data is a useful learning tool, so you will see another example later in this lesson, but in real applications you will rarely use this method because it can be clumsy and confusing. In practice, you will access the data via an event object inside an event handler.

result

If you're using an HTTPService defined as follows,

```
<s:HTTPService id="unitRPC"
    url="http://www.flexgrocer.com/units.xml"
    result="unitRPCResult(event)"/>
```

then the `unitRPCResult()` handler will be called when the XML is successfully retrieved. The proper method signature for this result handler is

```
private function unitRPCResult(event:ResultEvent):void{
}
```

You can access the unit node data in the body of the function by specifying

```
event.result.allUnits.unit
```

The returned data is available in the `result` property of the event.

To reiterate, the two ways of accessing data returned from this HTTPService are `unitRPC.lastResult`, which can be used anywhere in the application, or `event.result`, which can be used only inside the event handler.

Being Aware of Security Issues

The Internet is not a secure place. It's full of people "borrowing" information and attempting to access content they have no right to view. As Flash Player and your application must live in this world, many security restrictions are placed on what Flash Player is allowed to do on your system and how it is allowed to access data. The restriction that we are most concerned with in this section pertains to loading data from a remote server.

Flash Player uses a concept called *sandboxes* at the core of its security model. You can visualize this as a literal sandbox. All the children sitting in a single sandbox are allowed to interact and play with each other's toys. However, when a child from another sandbox wanders over to play or retrieve a toy, he is met with scrutiny and distrust.

Internally, Flash Player keeps all the content from different domains (mysite.com versus yoursite.com) in different sandboxes. As part of the security measures imposed on these sandboxes, content loaded from one domain is not allowed to interact with content loaded from another domain.


Following this logic, if your application is at <http://www.yoursite.com/yourApp.swf> and it attempts to load an XML file at <http://www.flexgrocer.com/units.xml>, it will be denied and a security error will occur as these two items exist in different security sandboxes.

The Flash Player security model requires the owner of flexgrocer.com to allow you access to that data, or you will be denied by default. The owner of that domain can allow such access by creating a *cross-domain policy file*. This file, named `crossdomain.xml`, specifies which domains have access to resources from Flash Player. The file is placed on the root of the web server that contains the data to be accessed. Here is an example of a cross-domain policy file that would enable your application on www.yoursite.com to access the `units.xml` file on flexgrocer.com. The file would reside in the web server root of flexgrocer.com:

```
<cross-domain-policy>
  <allow-access-from domain="www.yoursite.com"/>
</cross-domain-policy>
```

You can also use wildcards in a cross-domain policy file. This example allows anyone to access data:

```
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

 **TIP:** Browse the URL www.flexgrocer.com/crossdomain.xml to see the cross-domain file that allows you to retrieve data for this book. Also check www.cnn.com/crossdomain.xml to see who CNN allows to syndicate their content using Flash Player.

More information about the sandbox restrictions of Flash Player is available in the tech note on the Adobe site at www.adobe.com/cfusion/knowledgebase/index.cfm?id=tn_14213, along with a complete description of the cross-domain policy file, which can be found at www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.

Before deploying a cross-domain security file like this on a server, make sure you understand all the ramifications.

Retrieving XML Data via HTTPService

In this exercise, you will use an HTTPService object to retrieve data that contains the categories for grocery items—such as Dairy or Meat. You will also use the debugger to make sure that the data is returned correctly and will verify that you can see the data in the event object.

- 1 Open a web browser and go to the following URL:

<http://www.flexgrocer.com/category.xml>.

Notice the structure of the XML. This is the data you will retrieve using the HTTPService.

```
<?xml version="1.0" encoding="utf-8" ?>
<catalog>
  <category>
    <name>Dairy</name>
    <categoryID>4</categoryID>
  </category>
  <category>
    <name>Deli</name>
    <categoryID>5</categoryID>
  </category>
  <category>
    <name>Fruit</name>
    <categoryID>3</categoryID>
  </category>
  <category>
    <name>Meat</name>
    <categoryID>1</categoryID>
  </category>
  <category>
    <name>Seafood</name>
    <categoryID>6</categoryID>
  </category>
  <category>
    <name>Vegetables</name>
    <categoryID>2</categoryID>
  </category>
</catalog>
```

- 2 Open the FlexGrocer.mxml file.

Alternatively, if you didn't complete the previous exercise or your code is not functioning properly, you can import the FlexGrocer-PreHTTPService.fxp project from the Lesson06/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 3 Inside the `<fx:Declarations>` block, directly below the `<fx:XML.../>` tag, add an `<s:HTTPService>` tag. Give it an `id` of `categoryService`, and specify `http://www.flexgrocer.com/category.xml` as the `url` property. Specify a result handler named `handleCategoryResult` and be sure to pass the event object, as follows:

```
<s:HTTPService id="categoryService"
    url="http://www.flexgrocer.com/category.xml"
    result="handleCategoryResult(event)"/>
```

You are specifying the URL of the HTTPService to point to the XML you examined in step 1. In the next step, you will write an event handler with the name `handleCategoryResult()`, which will be called when the data has been retrieved.

- 4 In the Script block below the existing functions, add a new private function with the name `handleCategoryResult()` that returns `void`. The method will accept a parameter named `event`, typed as a `ResultEvent`. At this point the function is empty.

```
private function handleCategoryResult(event:ResultEvent):void{
}
```

If you chose `ResultEvent` in the pop-up list when you were typing, or if you pressed Enter when the `ResultEvent` was selected, then Flash Builder automatically added an import statement for you near the beginning of the Script block. If you do not see `import mx.rpc.events.ResultEvent;` near the beginning of your Script block, then you must add it manually now. Learning to use the code completion features of Flash Builder as soon as possible will save you hours in the course of this book and thousands of hours in your lifetime as a Flex developer.

- 5 Find the `handleCreationComplete()` method you wrote in Lesson 5, “Handling Events.” This method is called when the `creationComplete` event of the application is dispatched. Presently this method changes some data in the `groceryInventory` object. Delete the line that reads

```
groceryInventory.description = "Cheese from America's Dairyland";
```

from inside your method. You will now replace this with code to request data.

- 6 Call the `send()` method of the `categoryService` object inside the `handleCreationComplete()` method. Your method should read as follows:

```
private function handleCreationComplete(event:FlexEvent):void {
    categoryService.send();
}
```

The `categoryService` object is the HTTPService that you defined in your declarations. Invoking its `send()` method asks Flash Player to go out on the network and request the data found at `http://www.flexgrocer.com/category.xml`.

*** NOTE:** This is one of the most misunderstood aspects of programming in Flex. Data retrieval is asynchronous. Just because you called the `send()` method does not mean that your XML data is ready to use. The act of calling `send()` starts the process of getting that data. However, just as your web browser takes a moment to load a page, so this data will not be available and ready to use until the `result` event occurs.

- 7 Add a breakpoint on the closing bracket of the `handleCategoryResult()` method by double-clicking in the marker bar just to the left of the code and line numbers. A small blue dot will appear in the marker bar indicating the spot where program execution will halt.

Placing a breakpoint here gives you the chance to examine the data returned by the `HTTPService`.

- 8 Debug the application. Return to Flash Builder and make sure you are in the Debugging perspective. Double-click the Variables view tab. Drill down to the returned data by clicking the plus sign in front of `event > result > catalog > category`. Here you see the six category values in brackets [0], [1], [2], [3], [4], and [5] when you expand them.

If you dig far enough into the structure, you will eventually see categories such as Fruit, Meat, and Dairy.

Name	Value
event	mx.rpc.events.ResultEvent (@ec6c7d9)
[inherited]	
headers	null
_headers	null
result	mx.util.ObjectProxy (@ec6cf71)
catalog	mx.util.ObjectProxy (@ec6caf1)
category	mx.collections.ArrayCollection (@edd90c1)
[inherited]	
[0]	mx.util.ObjectProxy (@ec6cee1)
categoryID	4
dispatcher	flash.events.EventDispatcher (@ec6ce09)
_id	null
_item	Object (@edba101)
name	"Dairy"
notifiers	Object (@edba3d1)
object	Object (@edba101)
propertyList	null
proxyClass	mx.util.ObjectProxy (@d09ac41)
_proxyLevel	-1 [0xfffffff]
type	null
_type	null
uid	"E68B3A0B-EEE9-B553-EA51-DB96FBF632E5"
[1]	mx.util.ObjectProxy (@ec6cce9)

- 9 Double-click the Variables view tab to return it to its normal size. Terminate the debugging session by clicking the red Terminate button in the Debug or Console view. Finally, return to the Development perspective.

You have now used an `HTTPService` object to retrieve data, and you used debugging techniques to confirm that it has been returned to the application. Soon, you'll put the data to use.

Searching XML with E4X

In this section, you will gain some understanding of working with XML in Flex. ActionScript 3.0 contains native XML support in the form of ECMAScript for XML (E4X). This ECMA standard is designed to give ActionScript programmers access to XML in a straightforward way. E4X uses standard ActionScript syntax with which you should already be familiar, plus some new functionality specific to E4X.

▼ CAUTION! The XML class in ActionScript 3.0 is not the same as the XML class in ActionScript 2.0. That class has been renamed “XMLDocument” so that it does not conflict with the XML class now part of E4X. The old XML document class in ActionScript is not covered in this book. You shouldn’t need to use that class except when working with legacy projects.

In this task and through the rest of this lesson, you will use E4X functionality. The new E4X specification defines a set of classes and functionality for XML data. These classes and functionality are known collectively as E4X.

First, for a very basic, very quick review of XML terminology, examine the XML object as it would be defined in ActionScript:

```
private var groceryXML:XML = new XML();
groceryXML=
<catalog>
  <category name="vegetables">
    <product name="lettuce" cost="1.95">
      <unit>bag</unit>
      <desc>Cleaned and bagged</desc>
    </product>
    <product name="carrots" cost="2.95">
      <unit>pound</unit>
      <desc>Baby carrots, cleaned and peeled</desc>
    </product>
  </category>
  <category name="fruit">
    <product name="apples" cost="1.95">
      <unit>each</unit>
      <desc>Sweet Fuji</desc>
    </product>
    <berries>
      <product name="raspberries" cost="3.95">
        <unit>pint</unit>
        <desc>Firm and fresh</desc>
      </product>
```

```

        <product name="strawberries" cost="2.95">
            <unit>pint</unit>
            <desc>Deep red and juicy</desc>
        </product>
    </berries>
</category>
</catalog>;

```

The following statements describe the XML object, with the XML terminology italicized:

- The *root node* is catalog.
- There are two category *nodes*, or *elements*; for our purposes these will be synonyms.
- The product node has two *child nodes* (*children*), called unit and desc.
- The product node has two *attributes*, name and cost.

*** NOTE:** If you scrutinize the XML in more detail, you will also see a <berries> node with both berry-related products nested inside. This is done intentionally to show the power of the E4X operators in the examples to follow.

One last concept that you must understand before continuing is the difference between XML and an XMLList. Put simply, valid XML always has a single root node. An XMLList is a list of valid XML nodes without its own root node. For example:

```

<root>
  <node1/>
  <node2>
    <childNode/>
  </node2>
  <node3/>
</root>

```

represents valid XML. Further, each of the nodes is a valid piece of XML in and of itself. Conversely, the following structure:

```

<node1/>
<node2>
  <childNode/>
</node2>
<node3/>

```

does not have a single root node and is not valid XML. It is however, a list of valid XML nodes, and we refer to this construct as an XMLList. This XMLList has a length of 3 as there are three nodes immediately inside it. Finally, if we were to examine the following XML:

```

<node1/>

```

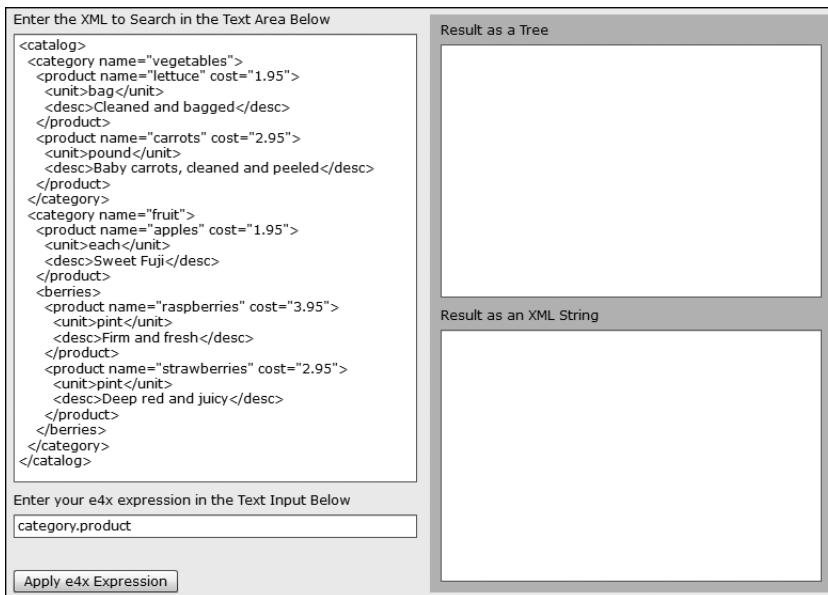
We could say this is valid XML, as it has a single root node, and it is a valid XMLList of length 1. All the E4X operators you are about to learn return XMLLists as their output type.

Now that you understand the basic XML terminology, you can start using some of the powerful E4X operators. A small application has been written for you to test some of these operators.

- 1 Import the E4XDemo.fxp project from Lesson06/independent folder into Flash Builder. Please refer to the appendix for complete instructions on importing a project.

A new project will appear in Flash Builder.

- 2 Inside the new project, open the E4XDemo.mxml file.
- 3 Run the E4XDemo application.



On the top left you see the XML shown earlier in this lesson. This application allows you to search that XML by entering an E4X expression into the text input on the bottom left and clicking the Apply e4x Expression button. The right side will display the resulting XMLList from that operation in two forms, as a tree of data on the top and as a formatted string on the bottom.

- 4 Click the Apply e4x Expression button to apply the default expression `category.product`.

*** NOTE:** In E4X expressions, the root node (in this case `catalog`) is part of the document and not used in statements.

This expression uses the *dot* (.) operator. This is one way to access data in the XML document. The dot operator behaves much like the dot in `object.property` notation with which you are familiar. You use the dot operator to navigate to child nodes. The expression yields the following results:

```
<product name="lettuce" cost="1.95">
  <unit>bag</unit>
  <desc>Cleaned and bagged</desc>
</product>
<product name="carrots" cost="2.95">
  <unit>pound</unit>
  <desc>Baby carrots, cleaned and peeled</desc>
</product>
<product name="apples" cost="1.95">
  <unit>each</unit>
  <desc>Sweet Fuji</desc>
</product>
```

In this case, the expression `category.product` indicates that you want all product nodes that are directly under category nodes. What was returned is an `XMLList`. Notice that the products that are children of the berries node did not appear.

Enter the XML to Search in the Text Area Below

```
<catalog>
<category name="vegetables">
<product name="lettuce" cost="1.95">
  <unit>bag</unit>
  <desc>Cleaned and bagged</desc>
</product>
<product name="carrots" cost="2.95">
  <unit>pound</unit>
  <desc>Baby carrots, cleaned and peeled</desc>
</product>
</category>
<category name="fruit">
<product name="apples" cost="1.95">
  <unit>each</unit>
  <desc>Sweet Fuji</desc>
</product>
<berries>
<product name="raspberries" cost="3.95">
  <unit>pint</unit>
  <desc>Firm and fresh</desc>
</product>
<product name="strawberries" cost="2.95">
  <unit>pint</unit>
  <desc>Deep red and juicy</desc>
</product>
</berries>
</category>
</catalog>
```

Enter your e4x expression in the Text Input Below

Apply e4x Expression

Result as a Tree

```
▶ product @name:lettuce @cost:1.95
▶ product @name:carrots @cost:2.95
▶ product @name:apples @cost:1.95
```

Result as an XML String

```
<product name="lettuce" cost="1.95">
  <unit>bag</unit>
  <desc>Cleaned and bagged</desc>
</product>
<product name="carrots" cost="2.95">
  <unit>pound</unit>
  <desc>Baby carrots, cleaned and peeled</desc>
</product>
<product name="apples" cost="1.95">
  <unit>each</unit>
  <desc>Sweet Fuji</desc>
</product>
```

- 5** Now enter the expression `category.product.unit` and click the button to apply it.

Here the dot operator again navigates the XML and returns the unit node for the three products retrieved in step 4.

```
<unit>bag</unit>
<unit>pound</unit>
<unit>each</unit>
```

- 6** Enter `category.product[1]` and apply the expression. This demonstrates that you can apply array notation in E4X. Here you get the second product because the list is zero indexed.

```
<product name="carrots" cost="2.95">
  <unit>pound</unit>
  <desc>Baby carrots, cleaned and peeled</desc>
</product>
```

This again shows that E4X lets you use familiar notation to work with XML. In previous versions of ActionScript, you had to use specific methods to access data in XML.

- 7** Enter `category.product.(unit=="bag")` and apply the expression. This limits the returned products to those whose unit node is bag. You've limited the data returned by putting a filter in the expression.

```
<product name="lettuce" cost="1.95">
  <unit>bag</unit>
  <desc>Cleaned and bagged</desc>
</product>
```

The parentheses implement what is referred to as *predicate filtering*.

- 8** Enter `category.product.(@cost=="1.95")` and apply the expression. Two product nodes are returned.

```
<product name="lettuce" cost="1.95">
  <unit>bag</unit>
  <desc>Cleaned and bagged</desc>
</product>
<product name="apples" cost="1.95">
  <unit>each</unit>
  <desc>Sweet Fuji</desc>
</product>
```

You have now performed predicate filtering on an attribute—hence the use of the attribute operator (`@`) in the parentheses (`@cost=="1.95"`). Also notice that if multiple nodes match the filter, you simply get multiple nodes returned—in this case both the lettuce and apples products.

- 9** Enter `category.product.(@cost=="1.95").(unit=="each")` and apply the expression. This expression demonstrates that you can apply predicate filtering multiple times. This results in only one product being returned.

```
<product name="apples" cost="1.95">
  <unit>each</unit>
  <desc>Sweet Fuji</desc>
</product>
```

- 10** Finally, to see the berry products get involved, enter `category..product` as the expression. You see that all products are returned, regardless of where they are in the XML.

```
<product name="lettuce" cost="1.95">
  <unit>bag</unit>
  <desc>Cleaned and bagged</desc>
</product>
<product name="carrots" cost="2.95">
  <unit>pound</unit>
  <desc>Baby carrots, cleaned and peeled</desc>
</product>
<product name="apples" cost="1.95">
  <unit>each</unit>
  <desc>Sweet Fuji</desc>
</product>
<product name="raspberries" cost="3.95">
  <unit>pint</unit>
  <desc>Firm and fresh</desc>
</product>
<product name="strawberries" cost="2.95">
  <unit>pint</unit>
  <desc>Deep red and juicy</desc>
</product>
```

This is an example of the very powerful *descendant* operator, represented by two dots (`..`). This operator navigates to the descendant nodes of an XML object, no matter how complex the XML's structure, and retrieves the matching nodes. In this case the descendant operator searched through the entire XML object and returned all the product nodes.

- 11** Enter `category..product.(@cost>2)` and apply the expression. This combines two operators and returns three products.

```
<product name="carrots" cost="2.95">
  <unit>pound</unit>
  <desc>Baby carrots, cleaned and peeled</desc>
</product>
<product name="raspberries" cost="3.95">
  <unit>pint</unit>
  <desc>Firm and fresh</desc>
```

```
</product>
<product name="strawberries" cost="2.95">
  <unit>pint</unit>
  <desc>Deep red and juicy</desc>
</product>
```

Here both predicate filtering and the descendant accessor are in use. E4X searched all the XML, regardless of position, and found three matches.

12 Close the E4XDemo project by right-clicking the project name and choosing Close Project.

You have now seen a slice of the very powerful E4X implementation in ActionScript 3.0. For more information, see “Working with XML” in the *Programming ActionScript 3.0* documentation that comes with Flex.

You can now return to the FlexGrocer project and begin working with dynamic XML.

Using Dynamic XML Data

As a Flex developer, you’ll have the opportunity to work with both XML and Objects. Over time you’ll decide which works better for your project in a specific situation. Starting with the next lesson and through the remainder of the book you will convert your XML to strongly typed objects, that is, objects with properties and methods defined well before the application executes. For the remainder of this lesson, however, you are going to work strictly with XML to display a list of categories.

While the techniques in this book are often presented in an improving fashion, meaning that those later in the book are often more-functional refactorings of earlier work, that is not the case with XML and Objects. These are simply two different techniques, each with advantages and disadvantages, that can be used to solve a problem.

As you saw in the previous exercise, XML is a quick and flexible way to present data. E4X expressions allow it to be searched and manipulated extremely quickly, and this makes it very powerful.

However, as you will no doubt discover, an application can fail due to a simple typographical error, something that typed objects can resolve at the expense of flexibility.

Currently the HTTPService tag in your FlexGrocer application is defaulting to returning dynamic Objects instead of XML when retrieving data. You are going to modify this property as well as store the returned data in an XMLListCollection for future use.

You will be working with the data retrieved from <http://www.flexgrocer.com/category.xml>. The structure of that XML file is listed here for your reference in this exercise.

```
<?xml version="1.0" encoding="utf-8" ?>
<catalog>
  <category>
    <name>Dairy</name>
    <categoryID>4</categoryID>
  </category>
  <category>
    <name>Deli</name>
    <categoryID>5</categoryID>
  </category>
  <category>
    <name>Fruit</name>
    <categoryID>3</categoryID>
  </category>
  <category>
    <name>Meat</name>
    <categoryID>1</categoryID>
  </category>
  <category>
    <name>Seafood</name>
    <categoryID>6</categoryID>
  </category>
  <category>
    <name>Vegetables</name>
    <categoryID>2</categoryID>
  </category>
</catalog>
```

1 Open the FlexGrocer.mxml file.

Alternatively, if you didn't complete the previous exercise or your code is not functioning properly, you can import the FlexGrocer-PreXMLCollection.fxp project from the Lesson06/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

2 Inside the `<fx:Declarations>` block, find the `<s:HTTPService>` tag. Add a new property to this tag named `resultFormat` and specify the value as `e4x`.

```
<s:HTTPService id="categoryService"
  url="http://www.flexgrocer.com/category.xml"
  resultFormat="e4x"
  result="handleCategoryResult(event)"/>
```

The `resultFormat` property tells the `HTTPService` how it should provide any data retrieved from this request. By default it returns data as dynamic Objects wrapped in

ObjectProxy instances. Changing this format to e4x instead provides you with XML that you can manipulate using E4X operators.

- 3 Make sure that you have a breakpoint set on the closing bracket of the `handleCategoryResult()` method.
- 4 Debug the application. Return to Flash Builder and make sure you are in the Debugging perspective. Double-click the Variables view tab. Drill down to the returned data by clicking the plus sign in front of `event > result > catalog`. Here you see the six category values all represented as XML. Expanding any of these nodes will provide you with more detail.

Name	Value
▶ this	FlexGrocer (@13e2f0a1)
▲ event	mx.rpc.events.ResultEvent (@1a...
▶ [inherited]	
▶ headers	null
▶ _headers	null
▶ result	XML
▶ <catalog>	
▶ <category>	
▶ <name>	"Dairy"
▶ <categoryID>	"4"
▶ <category>	
▶ <category>	
▶ <category>	
▶ <category>	
▶ <category>	
▶ <category>	
▶ _result	XML
▶ statusCode	200 [0xc8]
▶ _statusCode	200 [0xc8]

- 5 Double-click the Variables view tab to return it to its normal size. Terminate the debugging session by clicking the red Terminate button in the Debug or Console view. Finally, return to the Development perspective.
- 6 Near the top of your Script block, just under the import statements, add a new private variable named `categories` of type `XMLListCollection`. If you used code completion, Flash Builder has already imported the `XMLListCollection` for you. If you did not, then add an import for `mx.collections.XMLListCollection` before continuing.

`XMLListCollection` is a special class that holds and organizes `XMLLists`. It allows for those `XMLLists` to be sorted and filtered. You'll learn about collections in the next lesson.

- 7 Directly above the variable you just created, you are going to add a metadata tag to indicate that the variable is bindable. Type `[Bindable]` directly above the variable definition.

[Bindable]

```
private var categories:XMLListCollection;
```

The Bindable metadata tag tells Flex to watch this particular collection for changes. In the event of a change, the Flex framework should notify everyone using this data so they can update and refresh their display. You'll continue to learn about this powerful feature as you progress through the book.

- 8** Inside the `handleCategoryResult()` method, you need to instantiate a new `XMLListCollection` and assign it to the `categories` variable you just created.

```
private function handleCategoryResult( event:ResultEvent ):void {
    categories = new XMLListCollection();
}
```

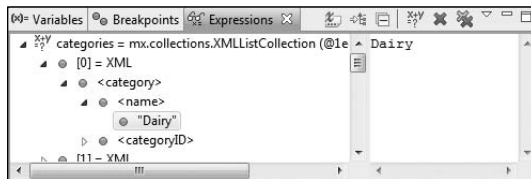
After this line of code executes, the `categories` variable will contain a new `XMLListCollection`. However, that collection does not yet contain your data.

- 9** Pass the E4X expression `event.result.category` into the constructor of the `XMLListCollection`.

```
private function handleCategoryResult( event:ResultEvent ):void {
    categories = new XMLListCollection( event.result.category );
}
```

This expression will return all the categories immediately inside the XML returned from the HTTPService call. By passing this to an `XMLListCollection` constructor, you are providing a way to further manage this data at runtime.

- 10** Make sure you have a breakpoint set on the closing bracket of the `handleCategoryResult()` method.
- 11** Debug the application. Return to Flash Builder and make sure you are in the Debugging perspective.
- 12** Select the word `categories` and then right-click it. Choose **Create Watch Expression**.
- Flash Builder will add `categories` to the Expressions view. If you cannot find the Expressions view, go to **Window > Expressions**.
- 13** Expand the `categories` object by clicking the triangle to the left of it in the Expressions view.



The Expressions view says that the type of item is an `mx.collections.XMLListCollection`. Inside the `XMLListCollection`, you'll find items denoted by array syntax. Expanding these items will reveal each of your categories.

- 14 Remove all the items from the Expressions view by clicking the Remove All Expressions button (the double X) to the right of the word *Expressions*.
- ▶ **TIP:** At any time, you may remove all the items from the Expressions view by clicking the double X or just a single item by highlighting it and clicking the X.
- 15 Terminate your debugging session by clicking the red Terminate button and remove your breakpoint before continuing.

Using the XMLListCollection in a Flex Control

Your application now retrieves data from an `HTTPService` and stores it as an `XMLListCollection`. However, presently the only way to ensure that the application is working is to use the debugger. In this exercise you will display the category data in a horizontal list across the top of the application.

- 1 Open the `FlexGrocer.mxml` file.

Alternatively, if you didn't complete the previous exercise or your code is not functioning properly, you can import the `FlexGrocer-PreList.fxp` project from the `Lesson06/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Add an `<s:List>` control inside the `controlBarContent` section of your `Application`. You can add this immediately after the existing `Buttons`.

```
<s:controlBarContent>
  <s:Button id="btnCheckout" label="Checkout" right="10" y="10"/>
  <s:Button id="btnCartView" label="View Cart" right="90" y="10" click.
State1="handleViewCartClick( event )"/>
  <s:Button label="Flex Grocer" x="5" y="5"/>
  <s:List>
</s:List>
</s:controlBarContent>
```

- 3 Specify that the `List` will remain 200 pixels from the left side of the `controlBar` and will have a height of 40 pixels.

```
<s:List left="200" height="40">
</s:List>
```


4 Specify that the List will use a `HorizontalLayout`.

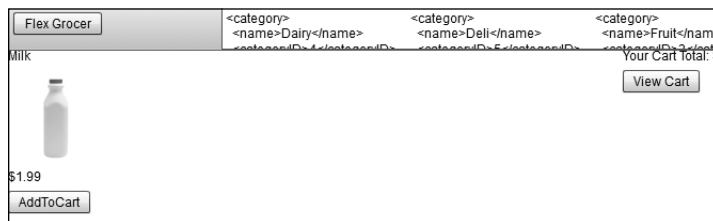
```
<s:List left="200" height="40">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
</s:List>
```

Previously you used horizontal and vertical layouts for groups, but `List` classes can also use these same layout objects to determine how their children should be arranged.

5 Now indicate that the `dataProvider` property of the `List` instance should be bound to the `categories` variable you defined and populated earlier.

```
<s:List left="200" height="40" dataProvider="{categories}">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
</s:List>
```

This syntax tells the Flex framework that, in the event the `categories` property changes, the list will need to be provided with the new value so that it can react. You will work extensively with `List` and `dataProvider` in future lessons.

6 Save and run the application.

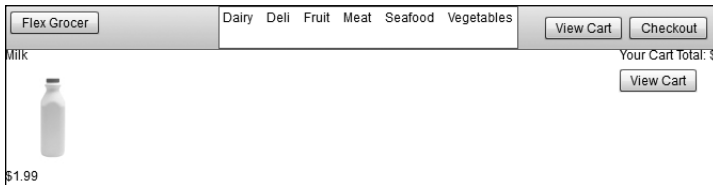
Your new list runs across the top of the page, with the elements arranged horizontally. Unfortunately, instead of displaying category names, you are now displaying the XML associated with the category. Notice that the data you really want displayed is in the `<name/>` node of the category XML.

7 Return to the `FlexGrocer` application and add a new property to your `List` called `labelField`. Set this property equal to `name`.

```
<s:List left="200" height="40" dataProvider="{categories}" labelField="name">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
</s:List>
```

The `labelField` property tells the list which field (property) inside your data to use as the label for the list item.

8 Save and run the application.



You now have a much more reasonable-looking list of category names that you will continue to use in the next lessons.

What You Have Learned

In this lesson, you have:

- Externalized your data as an XML file (pages 114–116)
- Used the external data first as an object and then as XML (pages 117–119)
- Used XML loaded at runtime (pages 119–123)
- Loaded remote XML data (pages 124–126)
- Learned about security sandboxes with remote data (pages 122–126)
- Explored E4X operators (pages 127–133)
- Used an `XMLListCollection` with your XML data (pages 134–137)
- Displayed your remote data in a Flex List (pages 137–139)

LESSON 7

What You Will Learn

In this lesson, you will:

- Create an ActionScript class to use as a value object
- Create ActionScript classes for a shopping cart
- Add functionality to the ShoppingCart and ShoppingCartItem classes

Approximate Time

This lesson takes approximately 1 hour and 30 minutes to complete.

LESSON 7

Creating Classes

Objects are the core of any object-oriented language. So far you have used classes provided for you by Adobe; however, to accomplish anything of even marginal complexity in Flex, you need to be comfortable creating your own. Objects are the realization of classes. Another way to state this is that a class is a blueprint for an object that will be created. In this lesson, you will first create several classes and then use them throughout the application.

```
1 package valueObjects {
2     [Bindable]
3     public class Product {
4         public var catID:Number;
5         public var prodName:String;
6         public var unitID:Number;
7         public var cost:Number;
8         public var listPrice:Number;
9         public var description:String;
10        public var isOrganic:Boolean;
11        public var isLowFat:Boolean;
12        public var imageName:String;
13
14        public function Product( catID:Number, prodName:String, unitID:Number, cost:Number, listPrice:Number, description
15            this.catID = catID;
16            this.prodName = prodName;
17            this.unitID = unitID;
18            this.cost = cost;
19            this.listPrice = listPrice;
20            this.description = description;
21            this.isOrganic = isOrganic;
22            this.isLowFat = isLowFat;
23            this.imageName = imageName;
24        }
25
26        public function toString():String {
27            return "[Product]" + this.prodName;
28        }
29    }
30 }
```

The finished Product data structure built in ActionScript 3.0 and integrated into the application

Building a Custom ActionScript Class

As mentioned at the end of Lesson 2, “Getting Started,” this book does not aspire to teach object-oriented programming (OOP), but every Flex developer needs at least a working knowledge of OOP terminology. So if you’re not familiar with terms like *class*, *object*, *property*, and *method*, now is a good time to take advantage of the hundreds, if not thousands, of OOP introductions around the web and in books.

You have already been building custom ActionScript classes in this book but may not have been aware of it because Flex initially hides this fact from you. When you build an application in MXML, you’re actually creating a new ActionScript class. Your MXML is combined with the ActionScript in the Script block, and a pure ActionScript class is created, which is then compiled into a SWF file for Flash Player. In the previous exercise, when you compiled FlexGrocer.mxml, a file named FlexGrocer-generated.as was created behind the scenes that contained the following code:

```
public class FlexGrocer extends spark.components.Application
```

You extended the Application class when you built FlexGrocer.mxml and Checkout.mxml. The same is true for every application you create using Flex.

▶ **TIP:** If you wish to see the ActionScript created, you can add a compiler argument in Flash Builder. Navigate to Project > Properties > Flex Compiler > Additional compiler arguments, and add `-keep-generated-actionscript` to the end of the existing arguments. A folder named `bin-debug/generated` will be created automatically in your project, and many ActionScript files will be placed there. Your application files will be in the form `Name-generated.as`. Don’t forget to remove the compiler argument when you’ve finished exploring.

In the first exercise of this lesson, you will build a class directly in ActionScript, without relying on Flex to convert MXML into ActionScript. Ultimately, this will give you much more granular control over your final code and encourage code reuse.

Building a Value Object

Value objects, also called data transfer objects (DTOs), or just transfer objects, are objects intended to hold data. Unlike other objects you've used so far, such as Labels and DataGrids, value objects are free from any logic other than storing and retrieving their data. These objects are implemented as ActionScript classes.

The name *data transfer object* comes from the fact that DTOs are often used for data transfer to the back end (server) of an application, often for permanent storage in a database. In this lesson, you will build a value object for a grocery product, along with objects for both a ShoppingCart and a ShoppingCartItem.

Before you get started, you need to understand the basics of building an ActionScript class. A very simple class is shown here, and labeled for discussion:

```
A package valueObjects.grocery {  
B public class Fruit {  
C   public var productName:String;  
  
D   public function Fruit() {  
       }  
  
E   public function toString():String {  
       return "Product " + this.productName;  
       }  
   }  
}
```

On line A, the *package* represents the path where the class is stored. In this example, you know the file is stored in a `valueObjects.grocery` package. On your hard drive, this means that the ActionScript file is stored in the `valueObjects/grocery` directory under your project.

On line B, the class is named `Fruit`. This is the name used to represent the class throughout an application (much like `DataGrid` or `Label`), and it must correspond to the name of the file. The `Fruit` class will be stored in a `Fruit.as` file on your drive.

On line C, the properties of the class are declared. This particular class has only a single public property, named `productName`, of type `String`. Multiple properties may be declared for any class.

Line D contains the constructor of the class. The constructor is called when a new object is instantiated from the class. The name of the constructor function must match the name of the class, which must match the name of the file. This function must be public, and it cannot have a return type listed.

In line E, the methods of the class are defined. This particular class has only a single method named `toString()`, but multiple methods may be declared.

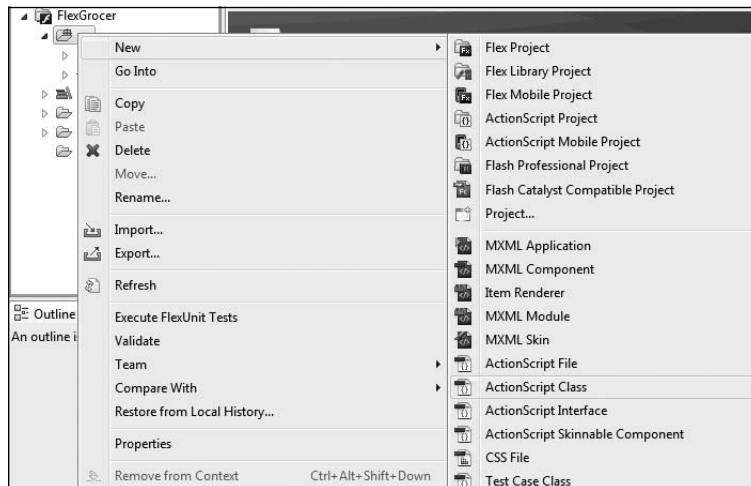
*** NOTE:** The terms *method* and *function* will often be used synonymously throughout the book. A function is a block of code that needs to be executed at some point in your application. A method is a function that belongs to a particular class, like the Fruit class here. In Flex it's possible to create functions and methods; however, every function you create in this book can also appropriately be called a method.

Throughout the FlexGrocer application, you will need to display and manage typed data and send this data to different objects in the application. In this exercise, you'll build a value object to hold information about a grocery product.

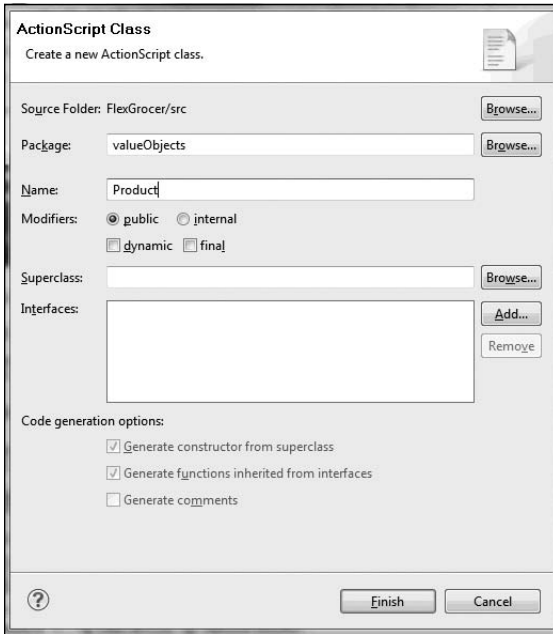
1 Open the FlexGrocer.mxml file that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson07/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

2 Create a new ActionScript class file by choosing File > New > ActionScript class.



3 Set the Package to **valueObjects** and the class Name to **Product**. Leave all other fields with the defaults. Click Finish to create the file.



First, this process created a package named `valueObjects` that you can now see in your Package Explorer. Next, it created a file named `Product.as` on your behalf. Finally, it populated that file with the required code for an ActionScript class.

Within the code, the words `package` and `class` are both keywords used in defining this class. Remember that this class will be a blueprint for many objects that you will use later to describe each grocery product.

- 4 In the `Product.as` file you need to add a `[Bindable]` metadata tag on the line between the package definition and the class statement.

```
package valueObjects {
    [Bindable]
    public class Product {
        public function Product() {
        }
    }
}
```


The `[Bindable]` metadata tag, when specified before the line with the `class` keyword, means that every property in this class can be used in data binding (that is, it can be monitored for updates by various Flex controls). Instead of specifying the whole class as `[Bindable]`, you can specify individual properties by locating the `[Bindable]` metadata tag over each property. For this application, you want every property in this class to be bindable.

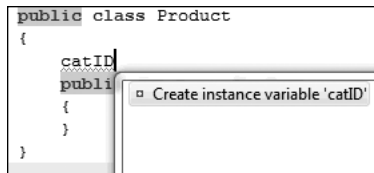
- 5 Inside the Product class definition, add a public property with the name `catID` and the type `Number`.

```
package valueObjects {
    [Bindable]
    public class Product {
        public var catID:Number;

        public function Product() {
        }
    }
}
```

All properties of a class must be specified inside the class definition in `ActionScript`.

- * NOTE:** If you were to type only `catID`, without anything else, you would see a question mark appear in an orange circle, indicating that it's not clear what you intended. Whenever you see that question mark in an orange circle , you can press `Control-1` on that line of code, which will launch the quick fix tool, offering potential solutions. On this line of code, clicking `Control-1` will offer you the option to create an instance variable from that line of code.



Choosing the option will add `private var` before the `catID`, and a datatype of `Object`. In this case, we want `catID` to be `public`, and datatyped as a `Number`, so it is less work to type it as originally intended, rather than use the quick fix feature.

- 6 Create additional public properties with the names `prodName` (`String`), `unitID` (`Number`), `cost` (`Number`), `listPrice` (`Number`), `description` (`String`), `isOrganic` (`Boolean`), `isLowFat` (`Boolean`), and `imageName` (`String`). Your class should appear as follows:

```
package valueObjects {
    [Bindable]
    public class Product {
        public var catID:Number;
        public var prodName:String;
        public var unitID:Number;
        public var cost:Number;
        public var listPrice:Number;
        public var description:String;
        public var isOrganic:Boolean;
        public var isLowFat:Boolean;
    }
}
```

```
public var imageName:String;

public function Product() {
}
}
```

You are creating a data structure to store inventory information for the grocery store. You have now created all the properties that will be used in the class.

When you created this class, Flash Builder created a default constructor on your behalf.

- 7 Edit this constructor to specify the parameters that need to be provided when a new instance of the Product class is created. These parameters will match the names and types of the properties you defined in the last step.

```
public function Product( catID:Number, prodName:String, unitID:Number,
↳ cost:Number, listPrice:Number, description:String, isOrganic:Boolean,
↳ isLowFat:Boolean, imageName:String ) {
}
```

The constructor function is called when an object is created from a class. You create an object from a class by using the `new` keyword and passing the class arguments. In this case the parameter names match the property names of the class. This was done to keep things clear, but is not necessary.

*** NOTE:** Two words are often used in discussions of methods: *parameter* and *argument*. They are often used interchangeably, but technically, functions are defined with parameters, and the values you pass are called arguments. So a function is defined to accept two parameters, but when you call it, you pass two arguments.

- 8 Inside the constructor, set each property of your object to the corresponding constructor parameter. When you are referring to the property of the class, you use the `this` keyword to avoid *name collision* (when the same name can refer to two separate variables).

```
\public function Product( catID:Number, prodName:String, unitID:Number,
↳ cost:Number, listPrice:Number, description:String, isOrganic:Boolean,
↳ isLowFat:Boolean, imageName:String ) {
    this.catID = catID;
    this.prodName = prodName;
    this.unitID = unitID;
    this.cost = cost;
    this.listPrice = listPrice;
    this.description = description;
    this.isOrganic = isOrganic;
    this.isLowFat = isLowFat;
    this.imageName = imageName;
}
```

This code will set each property of the object to the corresponding argument passed to the constructor. The first line of the constructor reads: “Set the `catID` property of this object to the value that was passed to the `catID` parameter of the constructor.”

- **TIP:** You could name the constructor parameters differently from the properties (for example, `categoryID` instead of `catID`). In that case, each property listed to the left of the equals sign could have done without the `this.` prefix (for example, `catID = categoryID;`). The prefix is added when you wish to be specific when referencing a property. The `this` prefix refers to the class itself and is implicit when there is no possibility of name collision.
- 9 Create a new method directly below the constructor function with the name `toString()` and the return type `String`. It will return the string `[Product]` and the name of the product. Your class should read as follows:

```
package valueObjects {
  [Bindable]
  public class Product {
    public var catID:Number;
    public var prodName:String;
    public var unitID:Number;
    public var cost:Number;
    public var listPrice:Number;
    public var description:String;
    public var isOrganic:Boolean;
    public var isLowFat:Boolean;
    public var imageName:String;

    public function Product( catID:Number, prodName:String,
      unitID:Number, cost:Number, listPrice:Number,
      description:String, isOrganic:Boolean, isLowFat:Boolean,
      imageName:String ) {
      this.catID = catID;
      this.prodName = prodName;
      this.unitID = unitID;
      this.cost = cost;
      this.listPrice = listPrice;
      this.description = description;
      this.isOrganic = isOrganic;
      this.isLowFat = isLowFat;
      this.imageName = imageName;
    }

    public function toString():String {
      return "[Product]" + this.prodName;
    }
  }
}
```

`toString()` is a special method of objects in ActionScript. Whenever you use an instance of your `Product` in a place where Flex needs to display a `String`, this method will be automatically invoked by Flash Player. A good example of this concept is the `trace()` method, which can be used to output data to the console. Running the code `trace (someProduct)` would call the `toString()` method of that `Product` instance and output the string it returns to the Console view. This can be very useful for debugging and displaying data structures.

- 10** Return to the `FlexGrocer.mxml` file and locate the `Script` block at the top of the page. Inside the `Script` block, declare a private variable named `theProduct` typed as a `Product`. Add a `[Bindable]` metadata tag above this single property.

```
[Bindable]
private var theProduct:Product;
```

If you used code completion, Flash Builder imported the `Product` class for you. If you did not, then add `import valueObjects.Product;` before continuing.

All MXML files ultimately compile to an ActionScript class. You must follow the same conventions when creating an MXML class as when creating an ActionScript class. For example, you must import any classes that are not native to the ActionScript language, such as the `Product` class you have built, and you must declare any properties that you will use in your MXML class.

- 11** Within the `handleCreationComplete()` method, but above the `categoryService.send();` statement, create a new instance of the `Product` class and assign it to the `theProduct` property. When creating the new `Product`, you will need to pass a value for each constructor argument. For these arguments you will use the data from the `<fx:XML>` tag named `groceryInventory`. Type the code as follows:

```
theProduct = new Product( groceryInventory.catID,
➤ groceryInventory.prodName, groceryInventory.unitID,
➤ groceryInventory.cost, groceryInventory.listPrice,
➤ groceryInventory.description, groceryInventory.isOrganic,
➤ groceryInventory.isLowFat, groceryInventory.imageName );
```

Here you are instantiating a new object of that `Product` class you built. You are passing the data from the `<fx:XML>` tag as constructor arguments. If you were to review the XML in the `groceryInventory` variable, you would note that it doesn't presently have a node for `catID` and `unitID`. However, in this context, Flash Player will just interpret them as 0 (zero) for the `Product` value object. These values will be used extensively when you begin loading more complicated data from the server.

*** NOTE:** When you're accessing properties from the `groceryInventory` XML, Flash Builder cannot help you with code completion or even compile-time checking. The nodes inside the XML aren't checked; this is why Flash Builder does not complain when you type `groceryInventory.catID` even though it is not present in the XML. This means it is extremely easy to make a typo that can be difficult to debug. For now check your code carefully, but as you continue to use strongly typed objects, you will see how Flash Builder can help and why typed objects can make debugging easier.

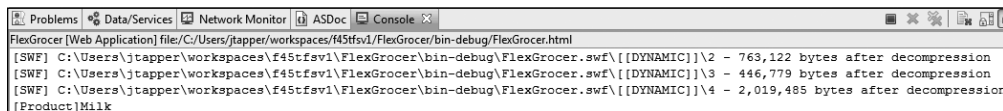
- 12** On the next line, add a `trace()` statement and trace the property `theProduct` out to the console. This statement will automatically execute the `toString()` method of your object and output the results. Your finished method should look like the following:

```
private function handleCreationComplete(event:FlexEvent):void {
    theProduct = new Product( groceryInventory.catID,
        ➤ groceryInventory.prodName, groceryInventory.unitID,
        ➤ groceryInventory.cost, groceryInventory.listPrice,
        ➤ groceryInventory.description, groceryInventory.isOrganic,
        ➤ groceryInventory.isLowFat, groceryInventory.imageName );

    trace( theProduct );
    categoryService.send();
}
```

- 13** Save and debug the application.

You should see `[Product]Milk` in the Console view, which indicates that you have created a `Product` value object successfully.



Building a Method to Create an Object

As you just did in the previous exercise, you can instantiate an instance of the `Product` class by passing values as arguments to the constructor. In this exercise, you'll build a method that will accept any type of object that contains all the properties and values needed for a product, and return an instance of the `Product` class populated with this data. This type of method is often referred to as a *factory* method, as its job is creating other objects.

Note that for this method to function correctly, the object passed to the method must contain property names that correspond exactly to the names you will hard-code into this method.

- 1 Be sure the `Product` class in the `valueObjects` package is open. Locate the `toString()` method. Immediately after this method, add the skeleton of a new `public static` method called `buildProduct()`. Be sure that the return type of the method is set to `Product`, and that it accepts a parameter named `o` typed as `Object`, as shown:

```
public static function buildProduct( o:Object ):Product {  
}
```

A *static* method is a method that belongs to a class, not to an instance. The methods you've worked with so far are called *instance* methods; they can be used only with instantiated objects.

Consider for a moment your `toString()` method. That method uses `productName` to display the name of the product represented by that object. If you have n product objects (where n is any number of product objects), each should display a different name when the `toString()` method is called. Since this method uses data from the object, it is logical that the object must exist before the method can be called.

Conversely, you may have a method that doesn't need (or care about) any of the data inside a specific instance. In fact, it could just be a utility method that does work independent of any particular instance. This is called a *static* method.

Static methods are often used for utilities such as the `buildProduct()` method. You will be able to call this method without creating an instance of the `Product` first. Used appropriately, static methods can increase the legibility and usefulness of your objects. To reference a static method with the name `buildSomething()` from the `Product` class, you would use the code `Product.buildSomething()`, which uses the class name before the method, as opposed to the instance.

- 2 Inside the `buildProduct()` method, create a new local variable named `p` of type `Product`.

```
var p:Product;
```

- 3 Below the local variable declaration, instantiate an instance of the `Product` class assigning it to `p` using the `new` keyword. Pass the `catID`, `prodName`, `unitID`, `cost`, `listPrice`, `description`, `isOrganic`, `isLowFat`, and `imageName` properties of the object `o` as arguments to the constructor. The `isOrganic` and `isLowFat` variables will be compared against the String `'true'` before being passed as the following code demonstrates:

```
p = new Product( o.catID, o.prodName, o.unitID, o.cost,  
➤ o.listPrice, o.description, ( o.isOrganic == 'true' ),  
➤ ( o.isLowFat == 'true' ), o.imageName );
```

Remember that the data used here is retrieved from the `<fx:XML>` tag. When data is retrieved this way, all the data is XML: The `true` and `false` values for these fields are just treated as a type of `String`. Comparing `isOrganic` and `isLowFat` to the String `'true'`

will return a Boolean value, either a `true` or a `false`. In this way, you are converting the value contained in the XML to the Boolean value that the newly created object is expecting for these properties. This allows you to take a string value from the XML, and convert it to a Boolean.

- 4 Return the object you just created by using the `return` keyword with the name of the object, `p`. Your final `buildProduct()` method should appear as follows:

```
public static function buildProduct( o:Object ):Product {
    var p:Product;

    p = new Product( o.catID, o.prodName, o.unitID, o.cost,
        ▶ o.listPrice, o.description, ( o.isOrganic == 'true' ),
        ▶ ( o.isLowFat == 'true' ), o.imageName );
    return p;
}
```

This method will create and return a new `Product` value object and populate it with data from the object passed as an argument.

- 5 Save the `Product.as` file.


The class file is saved with the new method. No errors should appear in the Problems view.

- 6 Return to `FlexGrocer.mxml`. In the `handleCreationComplete()` method, remove the code that builds `theProduct` and replace it with code that uses the static method to build `theProduct`. Remember to remove the `new` keyword.

```
theProduct = Product.buildProduct( groceryInventory );
```

This code calls the static method that builds an instance of the `Product` class, which returns a strongly typed `Product` value object from any type of object that has correspondingly named properties.

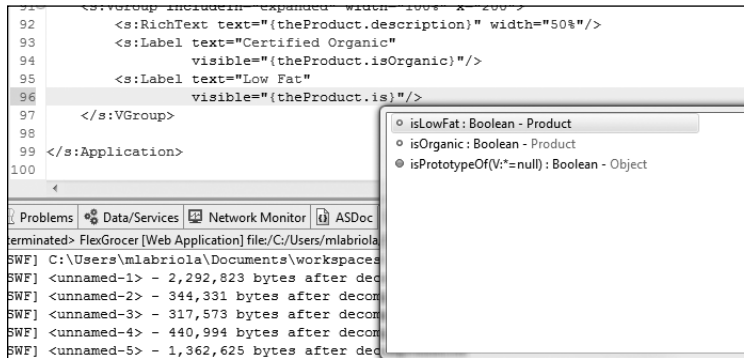
- 7 Locate the `VGroup` container, which, in the expanded state, displays the product description (whether the product is organic, and whether it is low fat). Change the `text` property of the `<s:RichText>` tag to reference the `description` property of the `theProduct` object you created in the `handleCreationComplete()` method. Also, add a `visible` property to both labels, and bind each to the appropriate `theProduct` object properties, as shown in the following code.

-  **TIP:** Remember, because `Product` is now an imported class, you can get code hinting for both the class name and its properties. When you are in the braces creating the binding, press `Ctrl-Spacebar` to get help for inserting the `Product` instance, `theProduct`. Then, after you enter the period, you will get the properties listed.

```

<s:VGroup includeIn="expanded" width="100%" x="200">
  <s:RichText text="{theProduct.description}" width="50%"/>
  <s:Label text="Certified Organic"
    visible="{theProduct.isOrganic}"/>
  <s:Label text="Low Fat"
    visible="{theProduct.isLowFat}"/>
</s:VGroup>

```



You are now referencing the value object you created. You also just gained one more benefit: Flash Builder is now helping you debug. If you were to make a typo—for example, if you were to type `theProduct.isLowerFat`—Flash Builder would alert you to the error when you saved. Now that Flash Builder knows the types of the objects you are using, it can verify that you are accessing properties that exist. What might have taken you a few minutes to reread, check, and perhaps even debug, Flash Builder found in moments. Over the course of a project, that becomes days and weeks of time.

8 Save and debug the application.

You should see that the `trace()` method performs just as before, and the correct data should still appear when you roll over the image.



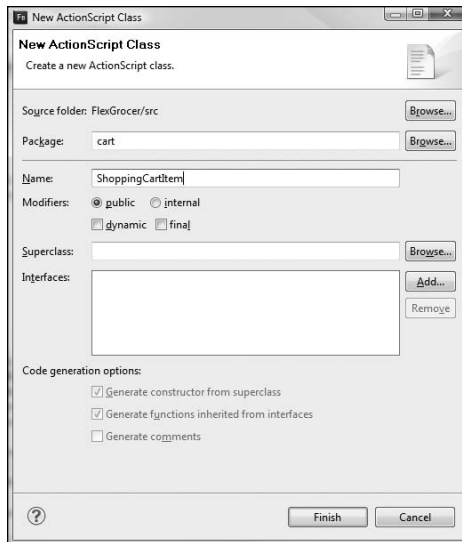
Building Shopping Cart Classes

In this exercise, you'll build a new class called `ShoppingCartItem`. This class will contain an item added to a shopping cart that you will also build shortly. The new class will keep track of a product added and its quantity. You will also build a method that calculates the subtotal for that item.

You will then build the skeleton for a `ShoppingCart` class that will handle all the logic for the shopping cart, including adding items to the cart.

- 1 Create a new ActionScript class file by choosing `File > New > ActionScript class`. Set the Package to `cart`, which automatically adds this class to a folder named `cart` inside your project. Enter `ShoppingCartItem` as the Name and leave all other fields with default values.

In this class, you will calculate the quantity of each unique item as well as the subtotal.



- 2 Within the class definition, define a public property with the name `product` and the type `Product`, as shown:

```
package cart {
    import valueObjects.Product;
    public class ShoppingCartItem {
        public var product:Product;

        public function ShoppingCartItem() {
        }
    }
}
```

The product is the most important piece of data in the `ShoppingCartItem`. If you used code completion, Flash Builder imported the `Product` class for you. If you did not, then add `import valueObjects.Product;` before continuing.

- 3 Define a public property with the name `quantity` of the type `uint`, as shown:

```
package cart {
    public class ShoppingCartItem {
        public var product:Product;
        public var quantity:uint;

        public function ShoppingCartItem() {
        }
    }
}
```

The data type `uint` means *unsigned integer*, which is a nonfractional, non-negative number (0, 1, 2, 3, ...). The quantity of an item added to the shopping cart will be either zero or a positive number, so `uint` is the perfect data type.

- 4 Define a public property with the name `subtotal` and the data type `Number`, as shown:

```
package cart {
    public class ShoppingCartItem {
        public var product:Product;
        public var quantity:uint;
        public var subtotal:Number;

        public function ShoppingCartItem() {
        }
    }
}
```

Each time a user adds an item to the shopping cart, you'll want the subtotal for that item to be updated. In this case, you are using `Number` as the data type. As the product's price is not likely to be an integer, the `Number` class allows for fractional numbers. Eventually, you will display this data in a visual control.

- 5 Edit the signature of the constructor of this class and specify the parameters that will be passed to this function. These parameters will include `product` typed as a `Product` and `quantity` typed as a `uint`. You will provide a default value of 1 for the quantity. If the developer calling this method does not provide a quantity, you will assume 1.

```
public function ShoppingCartItem( product:Product, quantity:uint=1 ) {
}
```

Remember that a constructor function must be public and that it never specifies a return type.

- 6 In the constructor, assign the object's properties to the values passed into the constructor's parameters. The names used are the same, so prefix the properties on the left side of the equal sign with `this`.

```
public function ShoppingCartItem( product:Product, quantity:uint=1 ){
    this.product = product;
    this.quantity = quantity;
}
```

Remember that the constructor is called every time an object is created from a class. The constructor will set the properties that are passed in—in this case, an instance of the `Product` class, and the quantity, which is set to 1 as a default. This object will be used only when an item is added to the shopping cart, so a default quantity of 1 seems reasonable.

- 7 Create a public method with the name `calculateSubtotal()` that will calculate the subtotal of each item by multiplying the `listPrice` of the product by the quantity, as follows:

```
public function calculateSubtotal():void{
    this.subtotal = product.listPrice * quantity;
}
```

When the user adds items to the shopping cart, you need to perform calculations so that the subtotal can be updated. Eventually, you also need to check whether the item has already been added to the cart; if so, you will update the quantity. You'll learn how to do this in the next lesson.

- 8 Call the `calculateSubtotal()` method on the last line of the constructor. This will ensure that the subtotal is correct as soon as the object is created.

```
public function ShoppingCartItem( product:Product, quantity:uint=1 ) {
    this.product = product;
    this.quantity = quantity;
    calculateSubtotal();
}
```

- 9 Create a public method with the name `toString()` that will return a nicely formatted string with the product's name and quantity. The returned string will read `[ShoppingCartItem]`, followed by a space, the product's name, a colon, and finally the quantity of that product in this `ShoppingCartItem`.

```
public function toString():String {
    return "[ShoppingCartItem] " + product.prodName + ":" + quantity;
}
```

As you learned previously, `toString()` methods are automatically called when Flash Player needs to represent this object as a `String`, such as when you use it in a `trace()` statement. This will provide you a lot of valuable debugging information.

- 10 Choose File > New > ActionScript class to create another new class. Set the package to **cart**. Name the class **ShoppingCart** and leave all other fields with default values.

Your new class will be the actual shopping cart, filled with `ShoppingCartItem` objects. This class will handle the manipulation of the data in the shopping cart. You have already created the visual look and feel of the shopping cart, and you will place all your business logic in this new class. This business logic includes work that must occur when adding an item to the cart, deleting an item from the cart, updating an item in the cart, and so on.

- 11 Create the skeleton of a public `addItem()` method, which returns `void`. The method will accept a parameter named `item`, of type `ShoppingCartItem`. In the method, add a trace statement that will trace the `item` added to the cart.

```
package cart {
    public class ShoppingCart {

        public function ShoppingCart() {
        }

        public function addItem( item:ShoppingCartItem ):void {
            trace( item );
        }
    }
}
```

This is the method in which you will add a new item to the shopping cart. You'll add more business logic to this method later. For now, you'll just trace the item added to the cart. Remember that the `toString()` method you wrote earlier is called automatically whenever an instance of the `ShoppingCartItem` class is traced.

- 12 Open `FlexGrocer.mxml` in Flash Builder and locate the Script block. Add an import for the `ShoppingCartItem` and `ShoppingCart` classes from the `cart` folder, as shown:

```
import cart.ShoppingCartItem;
import cart.ShoppingCart;
```

To use a class in a different package, your application needs an `import` statement that references the location or package in which the class is located.

- 13 After all the `import` statements, instantiate a public instance of the `ShoppingCart` class, name the instance `shoppingCart`, and add a `[Bindable]` metadata tag, as follows:

```
[Bindable]
public var shoppingCart:ShoppingCart = new ShoppingCart();
```

Pay attention to the differences in case here. Variables, such as `shoppingCart`, usually start with a lowercase letter. Classes, such as `ShoppingCart`, start with an uppercase letter.

When the user clicks the Add To Cart button, you want to call the `addItem()` method of the `ShoppingCart` class you just created. You will pass the `addItem()` method an instance of the `ShoppingCartItem` class. By instantiating the class here, you ensure that you have access to it throughout the application.

- 14** Locate the `handleViewCartClick()` method in the `<fx:Script>` block. Immediately after this method, add a new private function with the name `addToCart()` that returns `void`. Have the method accept a parameter named `product` typed as `Product`, as shown:

```
private function addToCart(product:Product):void {  
}
```

This method will be called when the user clicks the Add To Cart button, and you will pass an instance of the `Product` value object. As you do not intend anyone to call this method from outside this MXML class, you can use the private identifier. Using the keyword `private` here prevents others from calling this method unexpectedly.

- 15** Inside the `addToCart()` method, create a new instance of the `ShoppingCartItem` class with the name `sci` and pass the `product` parameter as an argument to the constructor.

```
private function addToCart( product:Product ):void {  
    var sci:ShoppingCartItem = new ShoppingCartItem( product );  
}
```

Notice that you passed the `product` but you did not pass the second parameter of the constructor (`quantity`). This is okay, as you provided a default value for the quantity when creating this class.

- 16** On the next line of the `addToCart()` method, call the `addItem()` method of the `shoppingCart` instance of the `ShoppingCart` class. Be sure to pass the `sci` object you just created to the method, as follows:

```
private function addToCart( product:Product ):void {  
    var sci:ShoppingCartItem = new ShoppingCartItem( product );  
    shoppingCart.addItem( sci );  
}
```

This code will call the `addItem()` method of the `ShoppingCart` class you built earlier. In the next sections, you'll learn how to loop through the data structure to see whether the item is added. For now, this method simply traces the name of the product added to the cart.

- 17** Find the Add To Cart button and add a handler for the `click` event that calls the `addToCart()` method, passing an instance of `theProduct`.

```
<s:Button label="Add To Cart" id="add"  
    click="addToCart( theProduct )"/>
```

Remember, the `addToCart()` method creates an instance of the `ShoppingCartItem` class and then passes that object to the shopping cart.

18 Save and debug the application.

Each time you click the Add To Cart button, you should see `[ShoppingCartItem] Milk:1` appear in the Console view.

```
[SWF] <unnamed-2> - 346,368 bytes after decompression
[SWF] <unnamed-3> - 317,573 bytes after decompression
[SWF] <unnamed-4> - 440,994 bytes after decompression
[SWF] <unnamed-5> - 1,362,625 bytes after decompression
[Product]Milk
[ShoppingCartItem] Milk:1
[ShoppingCartItem] Milk:1
```

Manipulating Shopping Cart Data

The next several exercises all deal with manipulating data in the shopping cart. You will work extensively with the Array class. In Lesson 8, “Using Data Binding and Collections,” you’ll add even more functionality to this class and allow it to work with Flex controls to update the display dynamically.

Adding Items to the Cart

In this exercise you’ll write the code to add items to the shopping cart.

1 Open the ShoppingCart.as file from the cart package that you used in the previous exercise.

Alternatively, if you didn’t complete the previous exercise or your code is not functioning properly, you can import the FlexGrocer-PreCartData.fxp project from the Lesson07/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

2 On the line after the class definition, define a public property with the name `items`, typed as an Array and set equal to a new Array instance.

```
package cart {
    public class ShoppingCart {
        public var items:Array = new Array();
```

This instantiates an Array object and assigns it to the `items` property. You will use this Array to track all the objects in the shopping cart.

3 Define a public property with the name `total`, typed as a Number. It will hold the total price of the items in this class. When the class is first instantiated, there won’t be any items, so set the value to 0 as shown:

```
public var total:Number=0;
```

Anytime a user adds an item to the cart, you will update this property with the price of the item. This will enable you to track the total cost of the end user’s order.

- 4 Locate the `addItem()` method of the `ShoppingCart` class and remove the `trace` statement. Use the `push()` method of the `Array` class to add `ShoppingCartItem` to the `items` array. The `push()` method of the `Array` adds an element to the end of the array, after any existing items.

```
public function addItem( item:ShoppingCartItem ):void {  
    items.push( item );  
}
```

In the previous exercise, you built a `ShoppingCartItem` class to hold data associated with items in a shopping cart. This class has properties to hold the product (an instance of the `Product` class), the quantity (an unsigned integer), and the subtotal (a number derived by multiplying the quantity by the price). When the user clicks the Add To Cart button, you pass a `ShoppingCartItem` to this method and place it in the `Array` using `addItem()`.

- 5 Create a public method with the name `toString()` that will return a nicely formatted string representing the items in the shopping cart. The returned string will be `[ShoppingCart`, followed by a space, a dollar sign, the closing right bracket, another space and the `items` array as shown:

```
public function toString():String {  
    return "[ShoppingCart $" + total + "]" + items;  
}
```

As you learned previously, `toString()` methods are called automatically when Flash Player needs to represent this object as a `String`, such as when you use it in a `trace()` statement. You added `toString()` methods to several of your objects already. When this statement executes, it is going to display `[ShoppingCart $0]` and then the array. The `Array` also has a `toString()` method, so when you trace this array, it will actually call the `toString()` method on each item in the array.

- 6 Switch back to `FlexGrocer.mxml` and locate the `addToCart()` method. To the last line of this method add a `trace()` statement that traces the `shoppingCart`.

```
private function addToCart( product:Product ):void {  
    var sci:ShoppingCartItem = new ShoppingCartItem( product );  
    shoppingCart.addItem( sci );  
    trace( shoppingCart );  
}
```

Each time you click the Add To Cart button, the entire shopping cart will be traced to the Console view.

- 7 Save and debug the application.

Each time you click the Add To Cart button, you will see another line appear in the Console view. For example, clicking it three times will yield:

```
[ShoppingCart $0] [ShoppingCartItem] Milk:1
[ShoppingCart $0] [ShoppingCartItem] Milk:1,[ShoppingCartItem] Milk:1
[ShoppingCart $0] [ShoppingCartItem] Milk:1,
[ShoppingCartItem] Milk:1,[ShoppingCartItem] Milk:1
```

The number of items grows each time another Milk product is added to the cart. However, this uncovers an error in your ShoppingCart. If you add Milk a second time, you likely mean to increase the quantity by 1, not actually add a new item. You will address this over the next few exercises.

Adding an Item or Updating the Quantity

The code you are about to write conditionally places a new item, or updates an existing item, in the shopping cart. It is not difficult line by line, but the logic involved can seem complex at first. To be sure you understand the big picture before you try to implement the details, let's walk through the logic required for the implementation.

1. The user clicks a button to add an item to the shopping cart.
2. The `addItem()` method is called.
3. The cart is checked to see if the item already exists.
 - If the item does not exist, it is added.
 - If the item does exist, you find and update the existing item.
4. The cart's total is updated.

Conditionally Adding a ShoppingCartItem

A key point in the logic of this exercise is determining whether a newly added `ShoppingCartItem` is already in the existing array of `ShoppingCartItems`. In this section, you will simply loop through the array looking for the correct item. In the next lesson, you'll learn to use collections and cursors to make this faster and more elegant.

Finding an Item in the Cart

The desired behavior is to have only new items added to the cart. If an item is clicked more than once, it should update the quantity of the item. You will need to write a method that can check whether an existing product is in the cart already.

- 1** In the `ShoppingCart.as` file, add a new private function named `getItemInCart()`. This method will accept a single parameter named `item` of type `ShoppingCartItem`. The method will return a `ShoppingCartItem`.

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {  
}
```

This method will accept a `ShoppingCartItem` and then look through all existing items to see if the represented product already exists in the cart.

- 2** On the first line of the `getItemInCart()` function, declare a new local variable named `existingItem` of type `ShoppingCartItem`.

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {  
    var existingItem:ShoppingCartItem;  
}
```

- 3** Directly below that variable declaration, create a `for` loop. Loops are blocks of code that are executed repeatedly using a series of values. In this loop, declare a variable `i` of type `uint` and loop from `0` to less than the length of the `items` array. In ActionScript, this loop appears like the following code:

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {  
    var existingItem:ShoppingCartItem;  
  
    for ( var i:uint=0; i<items.length; i++ ) {  
    }  
}
```

The code inside this `for` loop will be executed for each item in the array, allowing you to inspect each value for a matching product. The loop will continue to execute as long as `i` (the iterant) is less than the length of the `items` array. Each time the loop executes, the iterant is increased by 1 (`++` is shorthand for increment).

- 4** Inside the `for` loop, assign the local `existingItem` variable to the next item in the array. In ActionScript, the basic array contains data of an unknown type. This means that you will need to give the compiler a hint about the type of data in the array by casting it.

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {  
    var existingItem:ShoppingCartItem;  
  
    for ( var i:uint=0; i<items.length; i++ ) {  
        existingItem = items[ i ] as ShoppingCartItem;  
    }  
}
```

- 5 Still inside the loop, check whether `existingItem.product` is equal to `item.product`. If they are equal, return the `existingItem` variable. Finally, return `null` at the end of the method.

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {
    var existingItem:ShoppingCartItem;

    for ( var i:int=0; i<items.length; i++ ) {
        existingItem = items[ i ] as ShoppingCartItem;

        if ( existingItem.product == item.product ) {
            return existingItem;
        }
    }

    return null;
}
```

This code will now loop through the existing items and look for a matching product. If one is found, the code returns it. If a matching product is not found, `null` is returned.

Checking for an Item's Existence

While the `getItemInCart()` method allows you to find a given item in the cart, it would be nice to have a method that simply indicates whether or not the item is there; in other words, a method that returns a Boolean value indicating the item's existence.

- 1 Create a new private function named `isItemInCart()` that will return a Boolean. The method will accept a parameter named `item` of type `ShoppingCartItem`. Within the method, create a new variable local to the method with the name `sci`, which will hold a matched `ShoppingCartItem`, returned by the `getItemInCart()` method.

```
private function isItemInCart( item:ShoppingCartItem ):Boolean {
    var sci:ShoppingCartItem = getItemInCart( item );
}
```

The `getItemInCart()` method returns an `item` if it is found; else it will return `null`.

- 2 Add a return statement that returns the Boolean expression `(sci != null)`. Your completed method should look as follows:

```
private function isItemInCart( item:ShoppingCartItem ):Boolean {
    var sci:ShoppingCartItem = getItemInCart( item );

    return ( sci != null );
}
```

The expression `(sci != null)` will evaluate to either `true` or `false`; therefore, the `isItemInCart()` method will now return `true` if the added item is in the cart and `false` if the added item is not found in the cart.

Updating the Quantity of an Item Already in the Cart

The previous methods will help you to determine whether an item is in the cart. However, if that item is already in the cart, it shouldn't be added to the cart again. Rather, the item's quantity should be updated.

- 1 Create a skeleton for a new private `updateItem()` method, returning `void`. Have it accept a parameter named `item`, typed as `ShoppingCartItem`. On the first line of the method, define a local variable with the name `existingItem`, typed as a `ShoppingCartItem`. Set that local variable equal to the result of the `getItemInCart()` method, passing the `item`.

```
private function updateItem( item:ShoppingCartItem ):void {  
    var existingItem:ShoppingCartItem = getItemInCart( item );  
}
```

- 2 Still in the `updateItem()` method, update the `quantity` property of the `existingItem` object to its current value plus the value located in the `item` property.

```
existingItem.quantity += item.quantity;
```

Remember, whenever the Add To Cart button is clicked, a new item is added to the cart with the `quantity` value set to 1.

- 3 Still in the `updateItem()` method and immediately after you set the `quantity`, call the `calculateSubtotal()` method of the `existingItem` `ShoppingCartItem` instance. The final `updateItem()` method should look like this:

```
private function updateItem( item:ShoppingCartItem ):void {  
    var existingItem:ShoppingCartItem = getItemInCart( item );  
    existingItem.quantity += item.quantity;  
    existingItem.calculateSubtotal();  
}
```

When you first created the `ShoppingCartItem` class, you added a method with the name `calculateSubtotal()`, which updated a `subtotal` property with the `listPrice` of each product multiplied by the `quantity` of each product. Anytime you update the `quantity`, you need to recalculate that `subtotal` value.

- 4 Directly after the `updateItem()` method, create a skeleton for a private `calculateTotal()` method, with a return type `void`. In the method, create a local variable named `newTotal` of type `Number` and set it to an initial value of `0`. Create a second local variable named `existingItem` of type `ShoppingCartItem`.

```
private function calculateTotal():void{  
    var newTotal:Number = 0;  
    var existingItem:ShoppingCartItem;  
}
```

In this method, you will loop over the entire shopping cart and eventually update the `total` property of the `ShoppingCart` with the total of the user's items.

- 5 Still in the `calculateTotal()` method, create a skeleton of a `for` loop that will loop through the `items` array. Use the variable `i` as the iterant for the loop, with a data type `uint`. Use the `length` property of `items` as the terminating condition, and use the `++` operator to increment the iterant.

```
for ( var i:uint=0; i<items.length; i++ ) {  
}
```

Like the loop you wrote before, this enables you to loop through the entire shopping cart. The loop will continue to execute as long as `i` (the iterant) is less than the length of the array. Each time the loop executes, the iterant is increased by 1 (`++` is shorthand for this increase).

- 6 Inside the loop, set the `existingItem` variable equal to `items[i]` cast as a `ShoppingCartItem`. Update the `newTotal` variable with the `subtotal` of each existing item stored in the `items` array. Be sure to use the `+=` operator so it will add the new value to the existing one. Your `calculateTotal()` method should appear as follows:

```
private function calculateTotal():void{  
    var newTotal:Number = 0;  
    var existingItem:ShoppingCartItem;  
  
    for ( var i:uint=0; i<items.length; i++ ) {  
        existingItem = items[ i ] as ShoppingCartItem;  
        newTotal += existingItem.subtotal;  
    }  
}
```

This loops through the entire shopping cart and updates the `newTotal` variable by adding the `subtotal` (`price * quantity`) of each item in the cart to the current `newTotal`. Now anytime you need to calculate the total price of all the items, you can simply call this method.

- 7 As the final step of your `calculateTotal()` method, assign the value in the `newTotal` variable to the `total` property of the `ShoppingCart` instance.

```
private function calculateTotal():void{  
    var newTotal:Number = 0;  
    var existingItem:ShoppingCartItem;  
  
    for ( var i:uint=0; i<items.length; i++ ) {  
        existingItem = items[ i ] as ShoppingCartItem;  
        newTotal += existingItem.subtotal;  
    }  
  
    this.total = newTotal;  
}
```

- ▶ **TIP:** In the next lesson, you will tell the display to update each time the `total` property changes. By performing the calculation using the local `newTotal` property and then assigning it to the `total` at the end of the method, you will cause the display to update only once. Had you used the `total` property throughout, Flex would have tried to update the display once for every item in the array.

Checking Conditions in the `addItem()` Method

You now have all the building blocks to finish your `addItem()` method and ensure that the `total` remains consistent as items are added or updated.

- 1 Find the `addItem()` method. On the first line add an `if-else` statement. Use the `isItemInCart()` method to check whether the item is currently in the cart.

```
public function addItem( item:ShoppingCartItem ):void {
    if ( isItemInCart( item ) ) {
    } else {

        items.push( item );
    }
}
```

- 2 From within the `if` block, call the `updateItem()` method, passing the `item`. Move the code that pushes the new item into the `items` array into the `else` block.

```
public function addItem( item:ShoppingCartItem ):void {
    if ( isItemInCart( item ) ) {
        updateItem( item );
    } else {
        items.push( item );
    }
}
```

When the `addItem()` is called, it will check to see whether the item is already in the cart. If it is, then the existing item will be updated. Else, the new item will be added.

- 3 Finally, call the `calculateTotal()` method to recalculate the shopping cart's total after any add or update occurs.

```
public function addItem( item:ShoppingCartItem ):void {
    if ( isItemInCart( item ) ) {
        updateItem( item );
    } else {
        items.push( item );
    }

    calculateTotal();
}
```

4 Save and debug the application.

Each time you click the Add To Cart button, you will see another line appear in the Console view. For example, clicking it three times will yield:

```
[ShoppingCart $1.99] [ShoppingCartItem] Milk:1  
[ShoppingCart $3.98] [ShoppingCartItem] Milk:2  
[ShoppingCart $5.97] [ShoppingCartItem] Milk:3
```

Instead of adding a new item each time, the cart now updates the quantity of the item (as indicated by the last number). Also, note that the total in the shopping cart changes as you modify the cart. You will add more functionality, including the ability to remove items, in the next lesson.

What You Have Learned

In this lesson, you have:

- Created a Product class (pages 142–150)
- Created a static factory to build Product instances (pages 150–152)
- Populated a Product instance with data (pages 152–153)
- Created a ShoppingCartItem class (pages 154–156)
- Created a ShoppingCart class (pages 157–159)
- Manipulated shopping cart data (pages 159–161)
- Used an ActionScript loop to move through an array (pages 162–163)
- Added and updated items in the ShoppingCart (pages 164–167)

LESSON 8

What You Will Learn

In this lesson, you will:

- Learn how data binding works
- Replicate binding with event listeners
- Populate an ArrayCollection for binding
- Use an IViewCursor to locate, retrieve, and remove data in an ArrayCollection
- Hide internal functionality using implicit getters and setters

Approximate Time

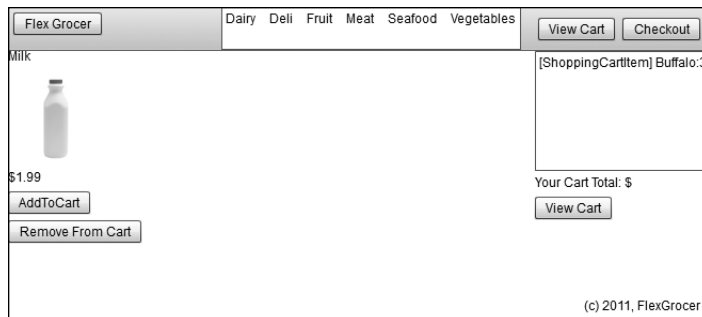
This lesson takes approximately 2 hours to complete.

LESSON 8

Using Data Binding and Collections

In the last lesson, you created a ShoppingCart class to hold items for purchase in your application. In this lesson, you'll continue building the ShoppingCart, as you enable it to work with a visual interface and learn to use the advanced features of the collection classes to manipulate and display up-to-date information to the user automatically.

The key to all these tasks resides in the Flex concept of data binding. So, you will start this lesson by learning how this important concept works, which will allow you to successfully apply it and determine when and where to use it appropriately.



The FlexGrocer application with a visual shopping cart

Examining Data Binding

Data binding is one of the key concepts that defines Flex. In fact it inspires a motto used in our daily consulting: *In Flex, the goal is to change the model (the data) and let the view (the components) follow.* In other words, you should try to avoid directly manipulating the components that make up the visual display at runtime and, instead, let those components react to changes you make to the data. This is the essence of data binding.

In Lesson 4, “Using Simple Controls,” you started using data binding as a method of updating the view automatically when the underlying data changed. Although you have used it periodically, you are now going to visit this concept in earnest.

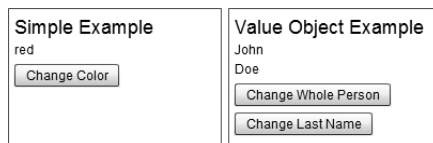
*** NOTE:** Flex 4.x also offers something called two-way binding. This is particularly useful with data input forms where you may want the data you type to update a variable. It will be addressed in Lesson 14, “Implementing the Checkout Process.”

Breaking a Simple Example

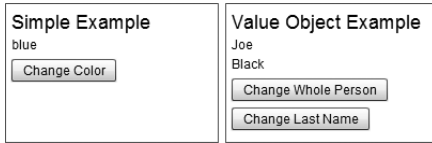
You will start by examining a simple example where data binding works to manipulate the view. You’ll then begin breaking it to see exactly when and where data binding ceases to work.

- 1 Import the `DataBindingLab.fxp` from the `Lesson08/independent` folder into Flash Builder. Please refer to the appendix for complete instructions on importing a project.
- 2 Open the `DataBindingLab.mxml` file and run it.

This is an extremely simple application designed to illustrate two use cases for data binding. Note that on startup the Simple Example shows the word `red` and the Value Object Example shows `John Doe` between the title and the buttons.



- 3 Click the `Change Color`, `Change Whole Person`, and `Change Last Name` buttons.



Not surprisingly, both examples change; let's examine why.

- 4 Close your web browser and return to Flash Builder. Find the lines of code that display the Simple Example and identify the property bound to the `colorName` Label.

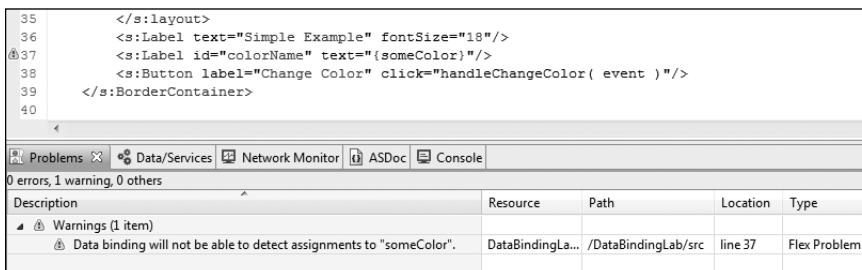
```
<s:BorderContainer width="200" height="130">
  <s:layout>
    <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
  </s:layout>
  <s:Label text="Simple Example" fontSize="18"/>
  <s:Label id="colorName" text="{someColor}"/>
  <s:Button label="Change Color" click="handleChangeColor( event )"/>
</s:BorderContainer>
```

Note that the `text` property of the `colorName` Label references the `someColor` property surrounded by curly brackets (braces) `{}`. You should read this line of code as, “The `text` property of the `colorName` Label is bound to `someColor`.” In this case, the braces `{}` are your way of telling Flex it should watch the `someColor` property for changes. If a change occurs, it should update the `colorName` Label with the new value.

- 5 Hold down the Control key (Command on the Mac) and click the word `someColor`.

This is a shortcut called *go to definition* in Flash Builder. It will move the cursor and focus to the location where that particular property is defined, even if it is in another class.

- 6 Remove the `[Bindable]` metadata tag from above the `someColor` property.
- 7 Save the file.



You will now see a warning symbol appear to the left of the line of code where the `someColor` property was being used. If you check your Problems view, you will also see a warning that says, *Data binding will not be able to detect assignments to “someColor”*.

The Flex compiler is once again trying to provide you with some valuable debugging information. It is letting you know that you have asked it (by using the braces) to watch the property `someColor`; however, it is not capable of doing so, as `someColor` is not bindable.

8 Debug the application.

Note that the word *red* appears on the screen at start-up. At this point it appears as though everything is still working correctly without the `[Bindable]` tag.

9 Click the Change Color button.

Regardless of how many times you click, the word *red* never changes to *blue* as it did previously.

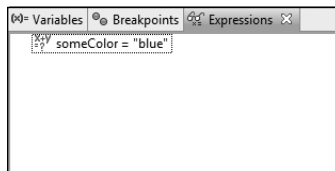
10 Leave the application running, but switch back to Flash Builder. Set a breakpoint on the closing brace of the `handleChangeColor()` method. This method is responsible for changing the `someColor` property from *red* to *blue*.

```
21 private function handleChangeColor( event:MouseEvent ):void {
22     someColor = "blue";
23 }
```

Remember, you can set or remove a breakpoint by double-clicking in the area to the left of the line numbers. You can do this before you debug the application or, as you are doing presently, while the application is running.

11 Return to the web browser and click the Change Color button. Flash Builder will stop at the breakpoint. Return to the Flash Builder Debug perspective.

12 Select `someColor`, right-click, and choose Create Watch Expression from the pop-up menu.



Even though the word *red* will continue to appear on the screen, the actual value of the `someColor` property is `blue`.

- ▶ **TIP:** At any time you may remove all the items from the Expressions panel by clicking the double X or remove just a single item by highlighting it and clicking the X.

The `[Bindable]` metadata tag that you place above a property like `someColor` allows Flex to watch that property for changes and react to that change by updating the view (visual components). Without the `[Bindable]` metadata tag, the property will still change when you assign it a new value; however, the view will not know to refresh.

- 13** Terminate the debug session.

Intentionally, you have now made the most common error that new Flex developers experience. Without the `[Bindable]` metadata tag, the word *red* still appears on start-up, but the Label fails to update later in the process. It is this behavior that the Flex compiler is attempting to bring to your attention via the warning: *Data binding will not be able to detect assignments to "someColor"*. Many developers ignore this warning and believe their application will work because the data appears correctly on start-up; unfortunately, however, the view will never update when the data changes.

Breaking a More Complicated Example

Having successfully broken a simple example, you are ready to break something more complicated. Previously you were working with the `someColor` property of the `DataBindingLab` application, which is a simple `String` type. When you use more complicated objects, data binding also becomes more complicated.

- 1** Make sure the `DataBindingLab.mxml` file is open, and examine the declaration of the `somePerson` property.

```
[Bindable]
private var somePerson:Person = new Person( "John", "Doe" );
```

At start-up the `somePerson` property is set to a new `Person` object, with the initial arguments `John` and `Doe` passed to the constructor. As you want Flex to notice changes to this `Person` instance, the `[Bindable]` metadata tag is also present above the declaration.

- 2** Hold down the Control (Command) key and click the class name `Person`.

You are again using the “go to definition” shortcut in Flash Builder. This time the definition you clicked was not a property, but rather a class, so Flash Builder opens the `Person` class for you.

```

package valueObjects {
    [Bindable]
    public class Person {
        public var firstName:String;
        public var lastName:String;

        public function Person( firstName:String="", lastName:String="" ) {
            this.firstName = firstName;
            this.lastName = lastName;
        }
    }
}

```

Note that the `Person` class is a simple value object, much like the `Product` value object you built in previous lessons. The `Person` class defines two properties, `firstName` and `lastName`, both of which are strings and defined as `public`.

The entire class is marked `[Bindable]`, meaning that both `firstName` and `lastName` can be watched by Flex for changes. As a reminder, marking the whole class `[Bindable]` is equivalent to marking each individual property with its own metadata tag.

Unlike the previous example with the `someColor` property, you have two `[Bindable]` metadata tags in this example. One tag is used in the `DataBindingLab.mxml` file, where the `somePerson` property is defined, and another is used on the top of the `Person` class. Each serves a different purpose.

3 Remove the `[Bindable]` metadata tag from the top of the `Person` class definition.

4 Save the file.

```

39     <s:Label id="colorName" text="{someColor}"/>
40     <s:Button label="Change Color" click="handleChangeColor( event )"/>
41 </s:BorderContainer>
42
43 <s:BorderContainer width="200" height="130">
44 <s:layout>
45     <s:VerticalLayout paddingTop="10" paddingLeft="5"/>
46 </s:layout>
47 <s:Label text="Value Object Example" fontSize="18"/>
48 <s:Label id="firstName" text="{somePerson.firstName}"/>
49 <s:Label id="lastName" text="{somePerson.lastName}"/>
50 <s:Button label="Change Whole Person" click="handleChangePerson( event )"/>
51 <s:Button label="Change Last Name" click="handleChangePersonName( event )"/>
52 </s:BorderContainer>
53
54 </s:Application>
55

```

Problems Data/Services Network Monitor ASDoc Console

0 errors, 3 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (3 items)				
Data binding will not be able to detect assignments to "firstName".	DataBindingLa...	/DataBindingLab/src	line 48	Flex Problem
Data binding will not be able to detect assignments to "lastName".	DataBindingLa...	/DataBindingLab/src	line 49	Flex Problem
Data binding will not be able to detect assignments to "someColor".	DataBindingLa...	/DataBindingLab/src	line 39	Flex Problem

If you look at the `DataBindingLab.mxml` class, you will see warning symbols appear to the left of the line of code where the `somePerson` property is used. If you check your Problems view, you will see two additional warnings: *Data binding will not be able to detect assignments to "firstName"* and *Data binding will not be able to detect assignments to "lastName"*.

The Flex compiler is once again trying to provide you with some valuable debugging information. It is letting you know that you have asked it (by using the braces) to watch the properties `firstName` and `lastName`. However, it is not capable of doing so, as neither is marked as bindable.

5 Debug the application and again click the Change Whole Person button.

You may be surprised by the initial result. When you click the Change Whole Person button, the person information does indeed change.

6 Click the Change Last Name button.

Regardless of how many times you click, the word *Smith* never changes to *Black* as it did previously. So, by removing the `[Bindable]` tag from the `Person` class, you broke one of the two cases being demonstrated for the complex object.

7 Terminate the debug session and return to Flash Builder.

8 Examine the method that is executed when you click the Change Last Name button. It is called `handleChangePersonName()`.

```
private function handleChangePersonName( event:MouseEvent ):void {
    somePerson.lastName = "Black";
}
```

When you executed this method, it changed the `lastName` property inside the `Person` object.

9 Now, examine the method that is executed when you click the Change Whole Person button. It is called `handleChangePerson()`.

```
private function handleChangePerson( event:MouseEvent ):void {
    somePerson = new Person( "Joe", "Smith" );
}
```

When you execute this method, it assigns a brand-new `Person` to the `somePerson` property.

You can think of these two cases in the following way: In one case, you're swapping the entire `Person` object out for a new `Person`. When you do that, you expect Flex to respond by showing you the new `Person` on the screen.

In the second example, you are not changing the whole `Person`, but rather changing something about the `Person`. In this case, you're changing the `Person`'s last name, though

the person remains the same. When you change something inside the Person object (like the last name), you also expect Flex to respond by updating the view.

These two use cases need two separate `[Bindable]` metadata tags, one to indicate that you want Flex to watch for the Person to change, and one to indicate that you want Flex to watch whether the properties of the existing Person change.

- 10 Finish breaking this example by removing the `[Bindable]` metadata tag from above the `somePerson` property, and save the file.

In the `DataBindingLab.mxml` class, you will see additional warning symbols to the left of the line of code where the `somePerson` property is used. If you check your Problems view, you'll also see two additional warnings, both saying, *Data binding will not be able to detect assignments to "somePerson"*. Flex is letting you know that this class is broken.

Description	Resource	Path	Location	Type
0 errors, 5 warnings, 0 others				
Warnings (5 items)				
Data binding will not be able to detect assignments to "firstName".	DataBindingLa...	/DataBindingLab/src	line 47	Flex Problem
Data binding will not be able to detect assignments to "lastName".	DataBindingLa...	/DataBindingLab/src	line 48	Flex Problem
Data binding will not be able to detect assignments to "someColor".	DataBindingLa...	/DataBindingLab/src	line 38	Flex Problem
Data binding will not be able to detect assignments to "somePerson".	DataBindingLa...	/DataBindingLab/src	line 47	Flex Problem
Data binding will not be able to detect assignments to "somePerson".	DataBindingLa...	/DataBindingLab/src	line 48	Flex Problem

- 11 Run the application and click buttons at will.

While the properties behind the scenes are changing values, the visible Flex components are unaware that these changes are occurring and cannot update.

Being the Compiler

As much fun as it was to spend a bit of time intentionally breaking code, it's time to be constructive again. You are going to fix the `someColor` property to ensure data binding works again. Although you could do this by simply adding back the `[Bindable]` metadata tag above the property, it will be significantly more interesting and informative to use this opportunity to gain an understanding of what the `[Bindable]` tag does on your behalf.

`[Bindable]` and the braces (`{}`) are among the many examples of code generation in Flex. Back in Lesson 7, you learned about a compiler option (`-keep-generated-actionscript`) that allowed you to see the ActionScript classes that the Flex compiler created from the MXML files you wrote.

The Flex compiler takes the MXML, along with any ActionScript code in the Script block, and creates completed ActionScript classes. This is an example of code generation, in that Flex generates ActionScript code based on the MXML you wrote. Any single line of MXML may very well become multiple lines of ActionScript code.

[Bindable] and the braces ({}) work in a very similar way. When Flex is compiling your code and sees a [Bindable] tag, it realizes that you intend Flex to watch that property for changes. It takes this opportunity to create some new code to make that possible. Similarly, when the compiler encounters the braces, it writes code to update your control when the data changes. In this exercise, you'll manually replicate some of that same code to gain an understanding of the binding mechanism.

Implicit Getters and Setters

The first concept employed by the Flex compiler during code generation is called an *implicit getter and setter*. Implicit getters and setters are a way to create a pair of functions that act like a property. You have already created many properties inside Flex classes:

```
public var someColor:String;
```

You access this property in your code by simple assignments or reads:

```
someColor = "blue";  
trace( someColor );
```

Implicit getters and setters allow you this same freedom of simply setting and reading values, but with the option of doing extra work when the property changes (for example, recalculating a subtotal when a quantity changes). Using an implicit getter and setter, this same code can be written in the following way:

```
private var _someColor:String;  
private function get someColor():String {  
    return _someColor;  
}  
  
private function set someColor( value:String ):void {  
    _someColor = value;  
    //do some extra interesting stuff  
}
```

Did you notice the underscore added to the beginning of the variable? You would continue to access this property in your code by simple assignments or reads:

```
someColor = "blue";  
trace( someColor );
```


The opportunity to do that *other stuff* is one key to making data binding work in Flex. Whenever you mark a variable [Bindable], either on its own or as part of a whole class of bindable properties, Flex changes that variable to a property defined as an implicit getter and setter.

- 1 Open the DataBindingLab.mxml file and find the declaration of the `someColor` property. Change the property name to prefix it with an underscore as follows:

```
private var someColor:String = "red";
```

In Flex, the underscore in front of a variable is often used to indicate that a private variable will have an implicit getter and setter that should be used to access this data.

- 2 Create a private getter function that will return the `_someColor` variable named `someColor`. Its return type will be `String`. A getter is created just like a function; however, the word `get` appears between the keyword `function` and the name of the function. A getter cannot accept any parameters.

```
private function get someColor():String {  
    return someColor;  
}
```

- 3 Create a private setter function that will accept a single parameter named `value` of type `String`. It will return `void`. When called, this method will set the `_someColor` variable to the `value` parameter. A setter is created just like a normal function; however, the word `set` appears between the keyword `function` and the name of the function. A setter always accepts a single parameter and returns nothing. By convention this parameter is named `value`.

```
private function set someColor( value:String ):void {  
    someColor = value;  
}
```

- 4 Save the file.

You should not see any errors in your Problems view, although you will continue to receive the data binding warnings for the moment. Note that you didn't change any of the code that accessed the `someColor` property. This code will still continue to function as it did before.

Event Dispatching and Listening

Data binding is just event dispatching and listening. In fact, armed with the knowledge learned in this lesson and in Lesson 5, "Handling Events," you already have almost enough information to replicate data binding on your own.

What follows is a brief introduction to dispatching events in ActionScript. You will learn this in much greater detail in Lesson 11, "Creating and Dispatching Events."

Listening to an Event

So far you have listened for events dispatched by Flex components using MXML. You've done this by specifying the name of a function that is called when the event occurs and then generally passing the event object. Listening to an event from ActionScript is a similar process, as illustrated by the following MXML example:

```
<s:Button label="A Perfect Button" id="someButton"
  click="doSomethingPlease( event )"/>
```

is written in ActionScript as:

```
someButton.addEventListener( "click", doSomethingPlease );
```

You are simply calling a method named `addEventListener()` on the `someButton` instance. You are passing two arguments, the name of the event that is important to you (`click`), and the name of the function you would like to call if that event occurs (`doSomethingPlease`).

Therefore, the general form of `addEventListener()` is:

```
objectThatDispatchesEvent.addEventListener( eventName, functionToCall );
```

Dispatching an Event

So far, you've relied on Flex controls to dispatch all events; however, it is quite easy to dispatch your own events as well. Here is an example of dispatching an event from ActionScript:

```
var event:Event = new Event( "myEvent" );
this.dispatchEvent( event );
```

First, you create a new `Event` object with the name `myEvent`. Then you simply call a method named `dispatchEvent()` on an object, passing `myEvent` as an argument. Often you will see this combined into a single statement.

Therefore the general form of `dispatchEvent()` is:

```
objectThatDispatchesEvent.dispatchEvent( new Event( eventName ) );
```

Data Binding as Events

In this section, you will fill in the missing pieces to make the `someColor` property work as it did before you began breaking it earlier in this lesson.

- 1 For the last line of the setter function for the `someColor` property, you will now dispatch a new `Event` called `someColorChanged`.

```
private function set someColor( value:String ):void {
    _someColor = value;
    this.dispatchEvent( new Event( "someColorChanged" ) );
}
```

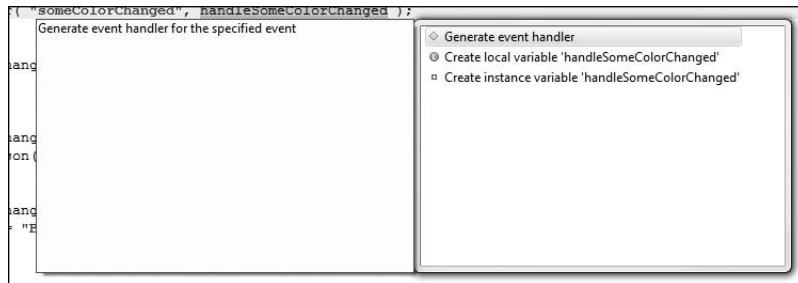
Now each time that the `someColor` property is set, you will call the `dispatchEvent()` method of the `Application` object (this refers to the `Application`, because the code is located in the `Application` Script block and runs in the `Application` context). Anyone listening for a `someColorChanged` event will be notified when it is dispatched.

- Find the `handleCreationComplete()` method of the `DataBindingLab.mxml` class. Here you will add an event listener to the `Application` for the `someColorChanged` event. When that event occurs, you will call a function named `handleSomeColorChanged()`.

```
private function handleCreationComplete( event:FlexEvent ):void {
    this.addEventListener( "someColorChanged", handleSomeColorChanged );
}
```

When the `someColor` setter dispatches its event, the `handleSomeColorChanged()` handler will be called. Notice the orange circle with a question mark on the left margin, as Flash Builder knows that `handleSomeColorChanged` is not yet a function in your class.

- Press `Control + 1` (Open `Apple + 1` on Mac) to launch the Quick Fix menu and choose *Generate event handler*.



Choosing this option instructs Flash Builder to create a new event handler method named `handleSomeColorChanged`. This new method will appear below the existing `handleChangePersonName()` method. The method will accept a single parameter named `event` of type `Event` and return nothing (`void`).

```
private function handleSomeColorChanged( event:Event ):void {
    // TODO Auto-generated method stub
}
```

- TIP:** When Quick Fix is first used, Flash Builder surrounds the method name in your event handler as well as the method name in the new function with a rectangle and highlighting. If you begin typing at that moment, Flash Builder will allow you to change the name of the event handler in all places it is being used.

- 4 Inside the `handleSomeColorChanged()` method, replace the auto-generated comment with ActionScript code to set the `text` property of the `colorName` Label equal to the value of `someColor`.

```
private function handleSomeColorChanged( event:Event ):void {  
    colorName.text = someColor;  
}
```

- 5 Find the `colorName` Label and remove the code that sets the `text` property.

```
<s:Label id="colorName"/>
```

You no longer need this code, as you are doing the work that the `[Bindable]` tag and the braces normally do on your behalf.

- 6 Directly below the `addEventListener()` call inside the `handleCreationComplete()` method, set the value of the `colorName.text` property to `someColor`.

```
private function handleCreationComplete( event:FlexEvent ):void {  
    this.addEventListener( "someColorChanged", handleSomeColorChanged );  
    colorName.text = someColor;  
}
```

Just as Flex set the initial value of your color regardless of any data binding later in the process, so you will set up your initial value here.

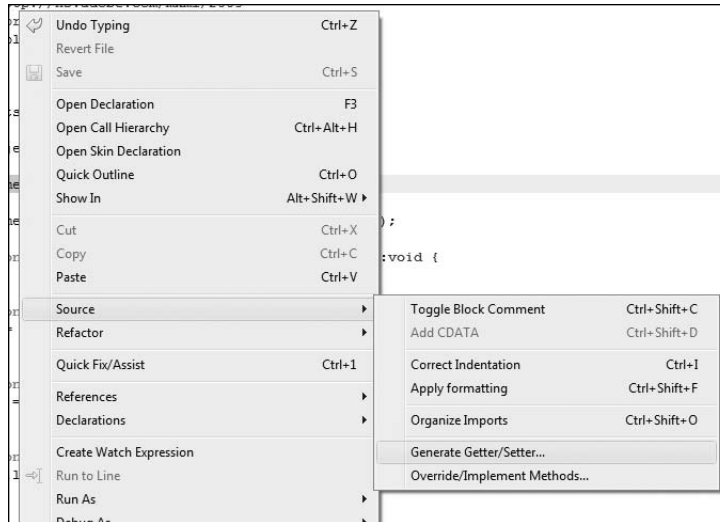
- 7 Run the application.

You will now be able to click the Change Color button again and see it change from *red* to *blue*.

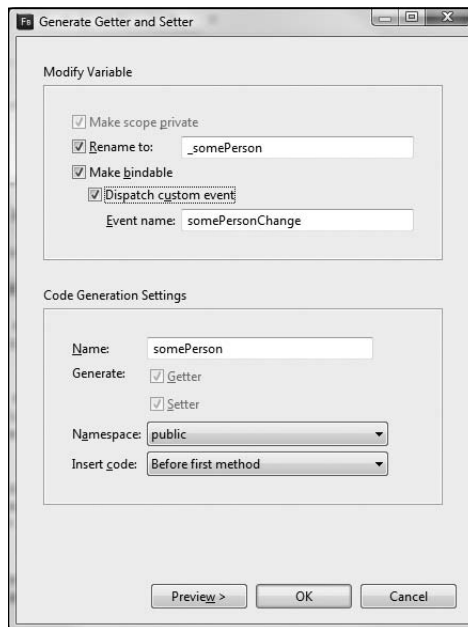
Using Auto-Generation with Flash Builder

So far in this lesson, you have learned about creating a getter and a setter and manually dispatching events. After you understand these concepts, you can apply shortcuts. In this section, you will accomplish the same goal, but allow Flash Builder to perform the balance of the work on your behalf.

- 1 Open the `DataBindingLab.mxml` file and find the declaration of the `somePerson` property. Right-click the property and from the menu choose `Source > Generate Getter/Setter`.



- 2 In the Generate Getter and Setter dialog box, select the Make bindable option and the Dispatch custom event option. Ensure your dialog box looks like the following image and click OK.



Flash Builder renames the `somePerson` property for you, creates the getter and setter, and dispatches an event when the value of `person` changes, much like you did manually in the last exercise. It also adds some logic to make sure that the value of `person` actually changes before dispatching the event and adds a special `Bindable` tag above the getter. This special tag allows this property to be used with the curly brackets. This is a very helpful shortcut to use during development.

```
private var _somePerson:Person = new Person( "John", "Doe" );
[Bindable(event="somePersonChange")]
public function get somePerson():Person {
    return _somePerson;
}
public function set somePerson(value:Person):void {
    if( _somePerson != value) {
        _somePerson = value;
        dispatchEvent(new Event("somePersonChange"));
    }
}
```

Understanding Bindable Implications

In the previous exercise you replicated a fair amount of the code that Flex writes each time you use braces and bindable metadata. The most important things to understand regarding this process are:

- Flex writes code on your behalf when you use these constructs.
- Data binding is just event dispatching and listening hidden behind the scenes.
- In order for data binding to work properly, objects must be able to dispatch events when something changes.

The last point has many implications. By default, not all classes in Flex can dispatch events. For example, the `Product` value object you created in Lesson 7 cannot dispatch events on its own. Classes that can dispatch events are called *event dispatchers*. In Flex all user interface components are a type of event dispatcher.

Fortunately, for classes like `Product`, when Flex sees the `[Bindable]` metadata tag, it changes your class during compilation to be an event dispatcher as well as generating all the code we discussed in this lesson.

It is important to remember that Flex does that work while it compiles your source code. However, this demonstrates an important prerequisite: Flex can do this work on the classes you write only where the source is present in your project. It cannot change classes that are already compiled and no longer exist as source. An important example of this is the `Array`.

The source code for the Array is not in your project; it exists inside Flash Player itself. That means that Flex has no capability of changing the code in the Array to dispatch events or add any of the code we have discussed. Therefore arrays cannot be used in data binding directly, as they cannot dispatch events that cause the user interface to update. In the next sections, you'll learn how Flex deals with these limitations through a technique called *proxying*.

Using ArrayCollection

In Lesson 7, you worked with ActionScript Array instances to store and retrieve shopping cart data. Throughout the book you have worked with various types of Array, XML, and Object instances. All three of these classes are built into the Flash Player itself; that is, you won't find an ActionScript class file that describes their behavior. They simply exist as part of the toolbox you have available when writing ActionScript code.

They form part of a fundamental set of types available to you in Flex that includes other common types such as Number, String, int, uint, and Boolean. However, unlike those simple types, Arrays, Objects, and XML are complex, meaning that they don't store simple values like a Number, but rather store more complex data and often have methods (such as the `push()` method of the Array) that can be called on the type.

In the previous exercise, you learned that Flex enables data binding on complex objects by manipulating the source code during compilation to allow objects to dispatch events. With these built-in types, such manipulation is not possible, and so another strategy must be used to allow their use in data binding. This strategy is called proxying. In Lesson 6, "Using Remote XML Data," you used two such proxies: an `XMLListCollection`, which was used so that your categories List would update when new data arrived from the server, and an `ObjectProxy`, which you observed when examining data retrieved from your `HTTPService`.

When used with data binding, a proxy's job is to act as a go-between for components you wish to be updated when a change occurs and a type, such as the Array, that does not have the proper logic to facilitate such an interchange.



Put simply, an `ObjectProxy` is a proxy for an `Object`, an `XMLListCollection` is a proxy for an `XMLList`, and an `ArrayCollection` is a proxy for an `Array`. This arrangement allows the use of these complex types with data binding.

In reality, the Array is fortunate to have two distinct proxies available, the ArrayList and the ArrayCollection. In this section, you'll learn about the ArrayCollection, as it not only provides the benefit of data binding but also has a rich set of additional features for sorting, filtering, and finding data quickly.

In the remainder of the book, you'll use ArrayList, as it is a simple and lightweight choice when you only need proxying capabilities.

Populating an ArrayCollection

In this exercise, you will create an ArrayCollection of Product objects, using a remote XML file for their source. This ArrayCollection of Product objects will represent the list of available products in your FlexGrocer store. You'll continue to use and manipulate this list through the remainder of the book.

- 1 Open a web browser and go to the following URL:

<http://www.flexgrocer.com/categorizedProducts.xml>

Notice the structure of the XML.

```
<?xml version="1.0" encoding="utf-8" ?>
<catalog>
  <category name="Meat" catName="Meat" catID="1">
    <product name="Buffalo"
      prodName="Buffalo"
      prodID="7"
      unitName="Pound"
      cost="4"
      listPrice="6.5"
      imageName="meat_buffalo.jpg"
      description="Delicious, low fat Buffalo sirloin. Better
        tasting than beef, and better for you too."
      isOrganic="No"
      isLowFat="Yes"
      unitID="3"
      catName="Meat"
      catID="1"/>
    <product name="T Bone Steak" .../>
    <product name="Whole Chicken" .../>
  </category>

  ...

</catalog>
```


Unlike the previous product data you used, the product nodes are listed beneath category nodes. Also, the critical information about the products is not described in nodes, but rather as attributes of the product node. You will need to use E4X operators to retrieve this data.

Table 8.1 Data Nodes and Attributes

Data as Nodes	Data as Attributes
<pre><product> <prodName>Milk</prodName> </product></pre>	<pre><product prodname="Milk"/></pre>

Finally, note that in our older XML, the values of the `isOrganic` and `isLowFat` nodes are represented by the words `true` or `false`. In this version, the words `No` or `Yes` have been substituted. This is typical of the real-world frustration of loading remote data from different sources. You'll learn how to deal with this change shortly.

2 Open the `FlexGrocer.mxml` file that you used in Lesson 7.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer.fxp` project from the `Lesson08/start` folder. Please refer to the appendix for complete instructions on importing a project should you ever skip a lesson or if you ever have a code issue you cannot resolve.

3 Inside `FlexGrocer.mxml`, below the `HTTPService` named `categoryService`, but still inside the `Declarations` block, add an `HTTPService` tag, with an `id` of `productService`. Set the `url` attribute of this tag to `http://www.flexgrocer.com/categorizedProducts.xml`.

```
<s:HTTPService id="productService"
  url="http://www.flexgrocer.com/categorizedProducts.xml"/>
```

4 Your new `HTTPService` should return its results as XML, so set the `resultFormat` to `e4x`. Also, specify that you will handle the `result` event of `HTTPService` with a new function named `handleProductResult()`, and pass the event object when it is called.

```
<s:HTTPService id="productService"
  url="http://www.flexgrocer.com/categorizedProducts.xml"
  resultFormat="e4x"
  result="handleProductResult(event)"/>
```

- 5 Ensure the cursor is over the word `handleProductResult` and press `Control+1`. Choose `Generate event handler`.

This will create a new protected function directly below the `handleCategoryResult()` function named `handleProductResult()`. The function will accept a single parameter named `event` of type `ResultEvent`, returning `void`.

```
private function handleProductResult( event:ResultEvent ):void {  
    // TODO Auto-generated method stub  
}
```

You will use this function to turn the data from the `HTTPService` into a series of `Product` objects.

- 6 Find the `handleCreationComplete()` method and delete the lines that build a new product from `groceryInventory` and the line that traces the `theProduct` variable.
- 7 Still inside the `handleCreationComplete()` method, add a call to `productService.send()` to retrieve your data from the server.

```
private function handleCreationComplete( event:FlexEvent ):void {  
    categoryService.send();  
    productService.send();  
}
```

Remember, simply creating the `HTTPService` tag does nothing to retrieve your data. You must call the `send()` method to issue the request for the `categorizedProducts.xml` file.

- 8 Save your application and set a breakpoint on the closing bracket of your new `handleProductResult()` function.

Remember you can set a breakpoint by double-clicking in the marker bar just to the left of the code and line numbers. A small blue dot will appear in the marker bar, indicating where the program execution will halt.

- ▶ **TIP:** Always make sure you save the application before attempting to set breakpoints. Setting breakpoints can be confusing and sometimes frustrating when the application is not yet saved.

- 9 Debug your application.

When you reach your breakpoint, return to Flash Builder and ensure you are in the `Debug` perspective.

- 10 Double-click the `Variables` view. Expand the event object and the `result` property. Further expand the `<catalog>` node beneath the `result` to ensure you are retrieving the correct data.

Name	Value
▶ this	FlexGrocer (@f28b0a1)
▲ event	mx.rpc.events.ResultEvent (@f3eb359)
▶ [inherited]	
▶ headers	null
▶ _headers	null
▶ result	XML
▶ <catalog>	
▶ <category name="Meat" catName="Meat" catID=	
▶ <product name="Buffalo" prodName="Buff	
▶ <product name="T Bone Steak" prodName=	
▶ <product name="Whole Chicken" prodNam	
▶ <category name="Vegetables" catName="Vege	
▶ <category name="Fruit" catName="Fruit" catID	
▶ <category name="Dairy" catName="Dairy" catID	
▶ <category name="Deli" catName="Deli" catID=	
▶ <category name="Seafood" catName="Seafooc	
▶ _result	XML
▶ statusCode	200 [0xc8]
▶ _statusCode	200 [0xc8]

You should see category nodes and, if you expand further, product nodes. Each product node will have a variety of attributes corresponding to the properties of your Product object.

11 Terminate your debugging session and return to the Flash perspective.

12 Open your Product value object class.

Previously, you created a static `buildProduct()` method that could build a Product from a generic object. Now you will create a new method that will create a Product from the attributes of XML.

13 Below the `buildProduct()` method, create a new public static method named `buildProductFromAttributes()`. This method will accept a single parameter named `data` of type XML. It will return a Product instance.

```
public static function buildProductFromAttributes( data:XML ):Product {
}
```

14 Immediately inside the method, create a local variable named `p` of type Product.

```
public static function buildProductFromAttributes( data:XML ):Product {
    var p:Product;
}
```

This variable will refer to your new Product instance. Next you will deal with the minor difference in the way the `isLowFat` and `isOrganic` nodes are handled in this XML file.

15 Now, create another local variable named `isOrganic` of type Boolean. Set it equal to an expression that checks whether `data@isOrganic` is equal to `Yes`.

```
var isOrganic:Boolean = ( data.@isOrganic == "Yes" );
```

This expression will check the attribute `isOrganic` against the String `Yes`. If they match, the variable `isOrganic` will be `true`.

- 16** Create a new local variable named `isLowFat` of type `Boolean`. Set it equal to an expression that checks whether `data@isLowFat` is equal to `Yes`.

```
var isLowFat:Boolean = ( data.@isLowFat == "Yes" );
```

- 17** Instantiate a new `Product` instance, passing the attributes from the `data XML` as the arguments of the `Product` constructor. In the case of the `isOrganic` and `isLowFat` nodes, pass the local `Boolean` variables instead. Finally return `p`, your new `Product` instance. Your code should read as follows:

```
public static function buildProductFromAttributes( data:XML ):Product {
    var p:Product;

    var isOrganic:Boolean = ( data.@isOrganic == "Yes" );
    var isLowFat:Boolean = ( data.@isLowFat == "Yes" );

    p = new Product( data.@catID,
        data.@prodName,
        data.@unitID,
        data.@cost,
        data.@listPrice,
        data.@description,
        isOrganic,
        isLowFat,
        data.@imageName );

    return p;
}
```

You now have three ways to create a new `Product`. You can call the constructor directly. You can call `buildProduct()` and pass an object or XML structure using nodes for the property names, or you can call `buildProductFromAttributes()` and pass it an XML structure with the properties as attributes. You will use the latter method shortly to make constructing your `ArrayCollection` much easier.

- 18** Return to the `FlexGrocer.mxml` file.
- 19** Find the `<fx:XML/>` tag with an `id` of `groceryInventory` and delete it.

As your data is now going to come directly from the server at runtime, you'll no longer need the local XML file.

- 20** Directly below the `categories XMLListCollection` in your `Script` block, add a new bindable private variable named `groceryInventory` of type `ArrayCollection`.

If you used code completion, the `ArrayCollection` will be imported for you. Otherwise, be sure to import `mx.collections.ArrayCollection`.

- 21** Return to your `handleProductResult()` method. Delete the auto-generated comment and create a new local variable named `products` of type `Array`. Set this variable equal to a new `Array` instance.

```
private function handleProductResult( event:ResultEvent ):void {
    var products:Array = new Array();
}
```

- 22** Below the `products` array, create another local variable named `resultData` of type `XMLList`. Set this variable to the E4X expression `event.result..product` as follows:

```
private function handleProductResult( event:ResultEvent ):void {
    var products:Array = new Array();
    var resultData:XMLList = event.result..product;
}
```

This E4X expression is referred to as a *descendant search*. As you learned in Lesson 6, you are indicating that you want all `<product>` nodes from the XML returned from the server, regardless of whether they are under other nodes (such as the category node in this case).

- 23** Insert a `for each..in` loop to move through each piece of XML in the `resultData` `XMLList`.

```
for each (var p:XML in resultData) {
}
```

The `for each..in` loop is similar to the `for` loop that you used previously. However, instead of a counter that moves from one number to the next over iterations, the `for each..in` loop understands items in a set and how to loop over them. In this case, the value of `p` will change at each loop to become the next product node in your `XMLList`.

- 24** Inside the `for each..in` loop, create a new local variable named `product` of type `Product`. Assign this variable to the result of the static method `buildProductFromAttributes()` on the `Product` class, passing it the variable `p`.

```
for each (var p:XML in resultData) {
    var product:Product = Product.buildProductFromAttributes( p );
}
```

This uses the new method you just created to build a typed `Product` object from the attributes of the XML node `p`.

- 25** Still inside the `for each..in` loop, use the `push()` method of the `products` array to add the newly created `Product` instance to the end of the `products` array.

```
for each (var p:XML in resultData) {
    var product:Product = Product.buildProductFromAttributes( p );
    products.push( product );
}
```

When your `for each..in` loop finishes executing, you will have an `Array` of `Product` objects that reflects the same data in your `XMLList` of product nodes.

- 26** Just below and outside the for each . . in loop, instantiate a new ArrayCollection, passing the products array as the constructor parameter. Assign the result to the groceryInventory property.

```
groceryInventory = new ArrayCollection( products );
```

In this example, you are passing the Array instance that the ArrayCollection will proxy to its constructor. Later in this lesson you will learn other ways to accomplish this same goal.

Your completed method should read as follows:

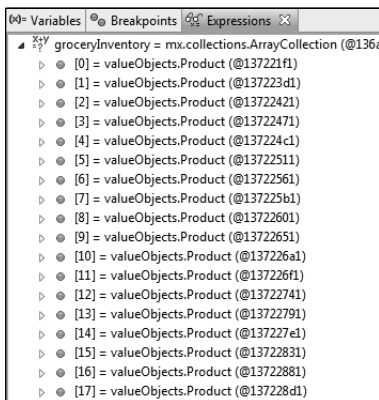
```
private function handleProductResult( event:ResultEvent ):void {
    var products:Array = new Array();
    var resultData:XMLList = event.result..product;

    for each (var p:XML in resultData) {
        var product:Product = Product.buildProductFromAttributes( p );
        products.push( product );
    }

    groceryInventory = new ArrayCollection( products );
}
```

This method will handle the result event from the HTTPService, and parse the returned XML, turning it into Product value objects. Those objects are then added to an ArrayCollection, where they can be used to update the user interface.

- 27** Save your application and debug it. When you encounter the breakpoint, switch to the Flash Debug perspective.
- 28** Add the groceryInventory property to your Expressions panel by highlighting it, right-clicking, and choosing Create Watch Expression. Expand the groceryInventory variable in the Expressions view, and you should see a list of Product objects.



- 29** Terminate your debugging session and return to Flash Builder. Remove your breakpoints.

Using Data from an ArrayCollection

In the previous exercise, you populated an ArrayCollection from XML data converted to objects. In this exercise, you'll use that data to populate the components in your view.

Data from an ArrayCollection can be accessed in several ways, as you will learn through the remainder of this lesson. Two of the most popular are via Array notation and via a special method of the ArrayCollection called `getItemAt()`.

The following statements will return the same data:

```
myArrayCollection[ 0 ];
myArrayCollection.getItemAt( 0 );
```

While these two statements are functionally equivalent, the call to `getItemAt()` has two distinct advantages. First, it is faster at runtime than the Array syntax, which exists primarily as a convenience to developers. Second, you can use `getItemAt()` with data binding to update your components at runtime.

- 1 Open the `FlexGrocer.mxml` file that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer-PreGetItem.fxp` project from the `Lesson08/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Find the Button instance with the label `AddToCart`. Presently, when that Button is clicked, you call the `addToCart()` method, passing it `theProduct`. Change the click handler to instead pass the data retrieved from calling the `getItemAt()` method of the `groceryInventory` collection, passing it a `0`. You will need to cast this data as a `Product` instance.

```
<s:Button label="Add To Cart" id="add"
  click="addToCart( groceryInventory.getItemAt( 0 ) as Product )"/>
```

Your application would be very boring if it displayed only one product, so we'll be adding multiple products in the near future. While this bit of code is certainly uglier than the code that was here before, it prepares your code for the important change from static to dynamic.

- 3 Find the RichText instance that uses the `description` property of the `theProduct` property. Change the `text` property to use the `description` property of the `groceryItem` collection's first item (index `0`).

```
<s:RichText
  text="{ groceryInventory.getItemAt( 0 ) as Product }.description}"
  width="50%"/>
```

This code, while still ugly, illustrates an important point. If the data inside the first position of the ArrayCollection were to change, this RichText instance's text property would update automatically. You will see that happen as you evolve the application in the upcoming lessons.

- 4 Update the *Certified Organic* and *Low Fat* Label instances in the same way, using the `getItemAt()` method.

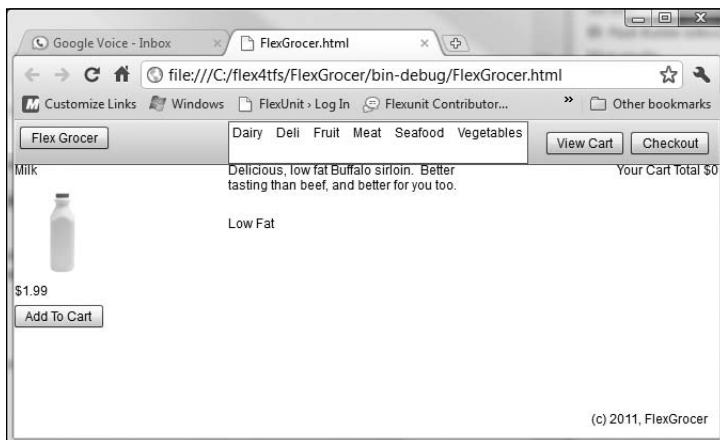
```
<s:VGroup includeIn="expanded" width="100%" x="200">
  <s:RichText
    text="{ groceryInventory.getItemAt( 0 ) as Product }.description}"
    width="50%"/>
  <s:Label text="Certified Organic"
    visible="{ groceryInventory.getItemAt( 0 ) as Product }.isOrganic"/>
  <s:Label text="Low Fat"
    visible="{ groceryInventory.getItemAt( 0 ) as Product }.isLowFat"/>
</s:VGroup>
```

- 5 Remove the `theProduct` variable declaration and the `[Bindable]` tag above it.

These are no longer needed because you are now referencing the collection directly.

- 6 Save and run your application.

If all the instances were changed correctly, the application should execute as before; however, when you hover over the bottle of milk, you should now receive the description and information for the first item in the `groceryInventory` collection, which happens to be Buffalo. You will continue to see the Milk bottle and the word *Milk*, as those are hard-coded in your application and will be changed in the next lesson.



Sorting Items in an ArrayCollection

In this lesson so far, you have used the ArrayCollection to allow you to make Array instances bindable. That is one of its most important uses; however, collections such as the ArrayCollection and XMLListCollection can do much more. In this exercise you'll replace the Array inside your ShoppingCart class with an ArrayCollection.

You will also use the sorting feature provided by the ArrayCollection to keep the items in your shopping cart in order at all times.

To sort an ArrayCollection, you'll use both the Sort and SortField classes. The following steps outline the process of sorting an ArrayCollection. You will implement these steps with more detail later in the task:

1. Create a new Sort object.
2. Create one or more SortField objects.
3. Assign the fields property of the Sort object an array of SortField objects (created in step 2).
4. Assign the Sort object to the sort property for the ArrayCollection.
5. Apply the sort by calling the refresh() method of the ArrayCollection.

Here is sample code that performs the steps to sort the items in an ArrayCollection.

```
var prodSort:Sort = new Sort();
var sortField:SortField = new SortField("someField");
prodSort.fields=new Array(sortField);
myArrayCollection.sort = prodSort;
myArrayCollection.refresh();
```

In the sample code, a SortField object was created to sort on the someField property of the objects in the collection. The constructor for the SortField object can take multiple arguments; however, only the first is required: the property name used while sorting. In this example the sort will use the someField property. Three other optional constructor parameters are available:

- Case sensitivity (false by default)
- Ascending versus descending (ascending by default)
- Numeric versus alphabetic (by default, Flex tries to guess if the data is numeric or alphabetic)

A single Sort object can have several sort fields (for example, you could sort first by category, then by price), which is why the fields property of the Sort class requires that an array of SortField instances to be specified. Even for a single-field sort, you create an array with only one SortField within it, as shown in the example.

- ▶ **TIP:** When specifying multiple SortFields, the order in the array is the order in which the sort fields would be applied. If you sort by category and then price, your code would look like this:

```
var prodSort:Sort = new Sort();
var sortField1:SortField = new SortField("category");
var sortField2:SortField = new SortField("listPrice");
prodSort.fields=new Array(sortField1, sortField2);
```

- 1 Open the ShoppingCart.as file that you built in the previous lesson.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer-PreSort.fxp project from the Lesson08/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Find the items array. Add a [Bindable] tag above this property and change the property's type to an ArrayCollection, assigning it to a new instance of the ArrayCollection without any constructor arguments.

```
[Bindable]
public var items:ArrayCollection = new ArrayCollection();
```

If you used code completion, the ArrayCollection will be imported for you. Otherwise, be sure to import `mx.collections.ArrayCollection`. As you learned previously, ArrayCollection proxy an Array. When you create a new ArrayCollection without specifying the Array instance to proxy, Flex creates a new Array on your behalf. In other words, these two lines are equivalent:

```
new ArrayCollection();
new ArrayCollection( new Array() );
```

- 3 Find the total property and add a [Bindable] tag above it.

```
[Bindable]
public var total:Number = 0;
```

You are allowing the view to watch this property and update if it changes.

- 4 In the constructor for the ShoppingCart class, create a new local variable named prodSort of type Sort. Set it equal to a new instance of the Sort class.

```
public function ShoppingCart() {
    var prodSort:Sort = new Sort();
}
```

If you used code completion, the Sort class will be imported for you. Otherwise, be sure to import `spark.collections.Sort`. The Sort class is used to define the order in which an ArrayCollection will keep its children.

There is also an `mx.collections.Sort` class. The difference is that the `spark.collections.Sort` class contains additional logic for sorting characters from multiple international locales. You should favor using the `spark.collections` classes when you have the choice.

- 5 After the `prodSort` variable, create another new local variable named `sortField` of type `SortField`. Set it equal to a new instance of the `SortField` class. Pass the string *product* to the `SortField` constructor.

```
public function ShoppingCart() {
    var prodSort:Sort = new Sort();
    var sortField:SortField = new SortField( "product" );
}
```

If you used code completion, the `SortField` class will be imported for you. Otherwise, be sure to import `spark.collections.SortField`. The `SortField` class is used to define various fields within your data structure that the `Sort` class will use when ordering.

- 6 After the `sortField` variable, you will set the `fields` property of the `prodSort` instance to an `Array` containing the `sortField`.

```
public function ShoppingCart() {
    var prodSort:Sort = new Sort();
    var sortField:SortField = new SortField( "product" );
    prodSort.fields = [ sortField ];
}
```

The square brackets in `ActionScript` are a shortcut to creating an `Array` instance. Here you are creating an array with one element: the `sortField`. The `fields` property accepts an array of `SortField` instances, so you may sort by multiple properties of the object.

*** NOTE:** When specifying multiple `SortFields`, the order of fields in the array is the order in which the sort fields are applied.

- 7 Set the `sort` property of the `items` `ArrayCollection` to the `prodSort` instance. Then call the `refresh()` method of the `items` `ArrayCollection`. Your constructor should look like the following code:

```
public function ShoppingCart() {
    var prodSort:Sort = new Sort();
    var sortField:SortField = new SortField( "product" );
    prodSort.fields = [ sortField ];

    items.sort = prodSort;
    items.refresh();
}
```

The `sort` property of the `ArrayCollection` references a `Sort` object that knows how to sort the collection. After applying a new sort to a collection, you must call the `refresh()` method to allow the sort to reorder its children and set up its internal state.

- 8** Find the `addItem()` method. Currently, when a new item is added to the cart, it is pushed onto the `items` array. However, `items` is now an `ArrayCollection`. Change the `push()` method of the `Array` to the `addItem()` method of the `ArrayCollection` instead.

```
public function addItem( item:ShoppingCartItem ):void {
    if ( isItemInCart( item ) ) {
        updateItem( item );
    } else {
        items.addItem( item );
    }

    calculateTotal();
}
```

The `addItem()` method will add the item to the collection and ensure that it stays properly sorted.

- 9** Switch to the `ShoppingCartItem` class. Add a `[Bindable]` metadata tag above the class definition for the `ShoppingCartItem`.

```
[Bindable]
public class ShoppingCartItem {
    public var product:Product;
    public var quantity:uint;
    public var subtotal:Number;
```

You want all the properties of the `ShoppingCartItem` to participate in data binding.

- 10** Switch to the `FlexGrocer.mxml` file and locate the `VGroup` named `cartGroup`.
- 11** Just above the `Label` with the text *Your Cart Total: \$*, add a new `<s:List/>` tag, with an `id` of `cartList`. Bind the `dataProvider` property of the `List` to the `shoppingCart.items` `ArrayCollection`. Finally, specify that this `List` will appear in `State1` using only the `includeIn` attribute.

```
<s:List id="cartList"
    dataProvider="{shoppingCart.items}" includeIn="State1"/>
```

This list will visually display the items in your `ShoppingCart` and update automatically thanks to data binding.

- 12** Save and run your application. As you click the `AddToCart` button repeatedly, *Buffalo* should initially appear and subsequently increment its item count.



Refactoring to Search with a Cursor

One of the features added to your shopping cart was the ability to determine whether a newly selected `ShoppingCartItem` already existed in the cart. Presently you are looping through items and doing a comparison to see whether that is the case.

In this exercise you're going to refactor the code responsible for that operation in the `ShoppingCart` to use a concept called a *cursor*. A cursor is a position indicator within the collection class that allows direct access to any particular item in the collection, allowing for the easy manipulation of items. Once you have a cursor created in a collection, you can:

- Move the cursor backward and forward
- Find specific items with the cursor
- Retrieve the item at the cursor location
- Add and remove items at the cursor position

All this functionality is available natively to the `ArrayCollection` class, meaning you do not need to write verbose loops to achieve any of these goals.

*** NOTE:** Cursors are not unique to `ArrayCollections`; they are available to several classes. For more information, read about the `IViewCursor` interface. For more information about interfaces in general, please refer to the "About Interfaces" section of the "Creating and Extending Flex Components" documentation.

The general steps to using a cursor in a collection class are:

1. Create a cursor for the collection using the `createCursor()` method.
2. Make sure the collection is sorted.

3. Use the `findFirst()`, `findAny()`, `moveNext()`, `movePrevious()`, and `seek()` methods to move the cursor and find items within the collection.

Now you will use the cursor while refactoring the `ShoppingCart` class.

- 1 Open the `ShoppingCart.as` class.
- 2 Find the `getItemInCart()` method and delete the for loop.

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {
    var existingItem:ShoppingCartItem;

    return null;
}
```

Going forward, you will use cursors to accomplish the same task.

- 3 Below the `existingItem` variable, create a new local variable named `cursor` of type `IViewCursor`. Set this variable equal to the result of calling the `createCursor()` method on the `items` `ArrayCollection`.

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {
    var existingItem:ShoppingCartItem;
    var cursor:IViewCursor = items.createCursor();

    return null;
}
```

If you used code completion, the `IViewCursor` interface will be imported for you.

Otherwise, be sure to import `import mx.collections.IViewCursor`. The `I` that prefaces the `IViewCursor` name indicates that it is an interface. The `createCursor()` method is not unlike the `buildProduct()` method you created earlier. It creates a new cursor, sets some initial values, and returns it for your use.

- * **NOTE:** Put simply, an interface is a contract between two objects. While you don't need to thoroughly understand interfaces to complete this section, you will need to understand the concept to use Flex effectively. There are many object-oriented references available online with great explanations and examples of interfaces.

- 4 After the call to `createCursor()`, pass the `item` parameter to the cursor's `findFirst()` method, and store the results in a Boolean variable named `found`.

```
var found:Boolean = cursor.findFirst(item);
```

In this step, you are using the `findFirst()` method of the cursor to search through the collection of `ShoppingCartItems` looking for a match. The `findFirst()` method expects an object as its argument. Flex uses the properties and values within that object to look

for a matching item. For example, the following code would search through a fictional collection of Flower objects looking at name properties.

```
var o:Object = new Object();
o.name = "Rose";
cursor.findFirst( o );
```

In this case, Flex notes a property called name in the object, and Rose as the value of that property. It then searches the collection for Rose. However, there's one very important point: You can search a collection only by the fields in your sort criteria. In your ShoppingCart, you created a sort based on the product field. So, even if you passed an object with hundreds of properties, Flex will compare only items in the product field.

If the findFirst() finds a match, the method will return a value of true, and the cursor will be positioned at the matching record. If no match is found, a value of false will be returned.

- ▶ **TIP:** In addition to findFirst(), the cursor also has the findAny() and findLast() methods. Any of these three could be used in the code because your logic prevents more than one ShoppingCartItem for each Product.

- 5 After the call to findFirst(), create an if statement that checks the found variable. If it is true, assign the cursor's current property to the existingItem variable, casting it as a ShoppingCartItem.

```
if ( found ){
    existingItem = cursor.current as ShoppingCartItem;
}
```

If findFirst() is successful, the current property of the cursor is a reference to the object at the present position of the cursor, which will be the ShoppingCartItem you just found. If the operation is not a success, this property is indeterminate and cannot be used safely.

- 6 Finally, change the return statement to return the existingItem. Your final method should look like this:

```
private function getItemInCart( item:ShoppingCartItem ):ShoppingCartItem {
    var existingItem:ShoppingCartItem;
    var cursor:IViewCursor = items.createCursor();

    var found:Boolean = cursor.findFirst( item );

    if ( found ){
        existingItem = cursor.current as ShoppingCartItem;
    }

    return existingItem;
}
```

Once a collection is sorted, the cursor's find methods are much faster, especially on large collections, than looping through the collection manually.

Removing Items with a Cursor

Your ShoppingCart is still missing one key feature, the ability to remove an item. You will add that ability now using cursor logic.

- 1 Open the ShoppingCart.as class.
- 2 Add a new public method just below addItem(), called removeItem(). The method will accept a single parameter named item of type ShoppingCartItem.
- 3 Create a new local variable within the removeItem() method named cursor of type IViewCursor, and assign the result of calling the createCursor() method on the items collection to it.

```
public function removeItem( item:ShoppingCartItem ):void {  
    var cursor:IViewCursor = items.createCursor();  
}
```

- 4 Create an if statement that evaluates whether a call to cursor.findFirst() passing item returns true.

```
public function removeItem( item:ShoppingCartItem ):void {  
    var cursor:IViewCursor = items.createCursor();  
  
    if ( cursor.findFirst( item ) ) {  
    }  
}
```

- 5 Inside the if block, call the cursor.remove() method.

This method removes the item at the cursor's current position.

- 6 Finally, call the calculateTotal() to re-total the cart after an item is removed. Your final method should look like this:

```
public function removeItem( item:ShoppingCartItem ):void {  
    var cursor:IViewCursor = items.createCursor();  
  
    if ( cursor.findFirst( item ) ) {  
        cursor.remove();  
    }  
  
    calculateTotal();  
}
```


- 7** Open FlexGrocer.mxml and find the Button with the label *Add To Cart*.

You will now add a Remove button.

- 8** Directly below this Button, add a new Button with the id of *remove*. Set the label to *Remove From Cart*.

Similar to what you did for the Add button, you will call a method when this button is clicked, passing it the current item.

- 9** On the `click` event of this Remove button, call a new method named `removeFromCart()`. Pass this new method the first (index 0) item from the `groceryInventory` collection, cast as a `Product`.

```
<s:Button label="Remove From Cart" id="remove"
  click="removeFromCart( groceryInventory.getItemAt( 0 ) as Product )"/>
```

- 10** Create a new private function named `removeFromCart()` directly below the `addToCart()` method in your Script block. The method will accept a single parameter named `product` of type `Product`.

- 11** Inside this method, create a new local variable named `sci` of type `ShoppingCartItem` and set it equal to a new instance of a `ShoppingCartItem`, passing `product` as the constructor argument.

- 12** As the last line of this method, call the `removeItem()` method of the `shoppingCart` instance and pass it `sci` as an argument.

```
private function removeFromCart( product:Product ):void {
  var sci:ShoppingCartItem = new ShoppingCartItem( product );
  shoppingCart.removeItem( sci );
}
```

- 13** Save and run your application. You now have the ability to add and remove items from the `ShoppingCart` with very little additional work, thanks to the cursor.

Filtering Items in an ArrayCollection

Collections provide one more crucial piece of functionality: *filtering*. Filtering provides a way for you to reduce the number of visible items in an `ArrayCollection` based on the results of a function.

Remember that an `ArrayCollection` is just a proxy to an `Array`. You already know that this proxy layer is useful in data binding, but it has other uses, namely, lying to you.

Each time you want a piece of data from the `Array`, you ask the `ArrayCollection`, which retrieves it for you. If you want to know how many items are in the `Array`, you ask the `ArrayCollection`, and it provides a length. However, what if the `ArrayCollection` is dishonest?

What if it reports fewer items in the Array than there really are? What if you ask for the item at position 3, and it returns the one at position 5 of the Array?

This seems extremely negative on the surface, but you have already used this lying behavior with success. When you sorted your ArrayCollection, the actual items in the Array (the data the ArrayCollection is proxying) remained unchanged. Instead, when you asked for the item at index number 2, the ArrayCollection simply returned what would be at index 2 *if* the Array were sorted in the way you requested.

Filtering is another very convenient way to lie. To filter an ArrayCollection you will need to implement these steps:

1. Create a new function that accepts a single parameter named `item`. This parameter will be the same type of whatever items are in your collection (for example, `Products`), or it can be generically of type `Object`. The function will return a `Boolean`, indicating whether the item should be included in the ArrayCollection's data.
2. Assign this function to the `filterFunction` property of the ArrayCollection.
3. Apply the function by calling the `refresh()` method of the ArrayCollection.

Here is sample code that performs the steps to filter the items in an ArrayCollection.

```
protected function filterOrganic( item:Product ):Boolean {  
    var includeMe:Boolean = item.isOrganic;  
    return includeMe;  
}
```

```
myArrayCollection.filterFunction = filterOrganic;  
myArrayCollection.refresh();
```

In the sample code, once `refresh()` is called, the ArrayCollection automatically passes each item in the Array to the `filterOrganic()` method. If this method returns a `true` (if the item is organic in this example), the ArrayCollection will continue to retrieve that item when asked. If the `filterOrganic()` method returns `false`, the ArrayCollection will decrement its length property by 1 and pretend that item never existed.

In all cases, the real data in the Array remains unchanged. This may seem overly complex, but it allows for a tremendous amount of functionality. Because the data in the Array remains unchanged, you can simultaneously see the data sorted or filtered in multiple ways, all using the same source Array. You will use this functionality in the coming lessons to filter your products by the category selected in the application control bar.

Refactoring ShoppingCartItem

With the new information learned in this lesson, one more piece of refactoring should occur. Right now, each time you change the quantity of a `ShoppingCartItem`, you also manually call `calculateSubtotal()`.

```
private function updateItem( item:ShoppingCartItem ):void {
    var existingItem:ShoppingCartItem = getItemInCart( item );
    existingItem.quantity += item.quantity;
    existingItem.calculateSubtotal();
}
```

In object-oriented programming, you strive to hide the internal workings of objects from the end user. Here, the internal workings are painfully obvious. Using the implicit getter and setter logic learned in this lesson, you can correct this issue.

- 1 Open the `ShoppingCart.as` class.
- 2 Find the `updateItem()` method and remove the call to `calculateSubtotal()` on the `existingItem`.

The `ShoppingCart` will no longer be responsible for executing this internal logic of the `ShoppingCartItem` class.

- 3 Open the `ShoppingCartItem.as` class.
- 4 Click the public variable named `quantity` and press `Control + I`. Choose to Create getter and setter for '`quantity`'. Leave all options at default.

The `quantity` variable will be renamed to `_quantity` and a getter and setter will be created:

```
private var _quantity:uint;
public function get quantity():uint {
    return _quantity;
}
public function set quantity(value:uint):void {
    _quantity = value;
}
```

- 5 Inside the setter, after the `_quantity` variable is set, call the `calculateSubtotal()` method.

```
public function set quantity( value:uint ):void {
    _quantity = value;
    calculateSubtotal();
}
```

Now, anytime someone sets the `quantity`, the `ShoppingCartItem` will automatically recalculate its subtotal.

- 6 As the last step, and to reinforce this point of encapsulating (hiding) internals, change the `calculateSubtotal()` method from `public` to `private`.

```
private function calculateSubtotal():void {  
    this.subtotal = product.listPrice * quantity  
}
```

Now code outside this class will be unable to call this method directly.

- 7 Save and run your code.

As with any refactoring, the code execution should be identical, with the ability to add, update, and delete shopping cart items.

What You Have Learned

In this lesson, you have:

- Learned how data binding works and common mistakes that cause it to cease to function (pages 170–176)
- Replicated data binding with event listeners (pages 176–184)
- Programmatically added items to an `ArrayCollection` built from remote XML data (pages 184–191)
- Used the `getItemAt()` method of the `ArrayCollection` to retrieve data (pages 192–193)
- Sorted an `ArrayCollection` (pages 194–198)
- Used a cursor to find and remove data (pages 198–202)
- Refactored `ShoppingCartItem` (pages 204–205)

LESSON 9

What You Will Learn

In this lesson, you will:

- Understand the need for components and how they can fit into an application architecture
- Understand the Flex class hierarchy
- Build both visual and non-visual components
- Instantiate and use custom components
- Create properties and methods in custom components

Approximate Time

This lesson takes approximately 2 hours to complete.

LESSON 9

Breaking the Application into Components

You have used many components while building the application to its current state. Every time you use an MXML tag, you are using a component. In fact, Flex is considered to be a component-based development model. In this lesson, you'll learn how to create your own components. The custom components you build will either extend functionality of the components that the Flex SDK provides or group functionality of several of those components together.

Up to this point, you did not have a way to divide your application into different files. If you continued this way, the application file would get longer and longer and become more difficult to build, debug, and maintain. It would also be very difficult for a team to work on one large application page. Components let you divide the application into modules, which you can develop and maintain separately. With careful planning, these components can become a reusable suite of application functionality.

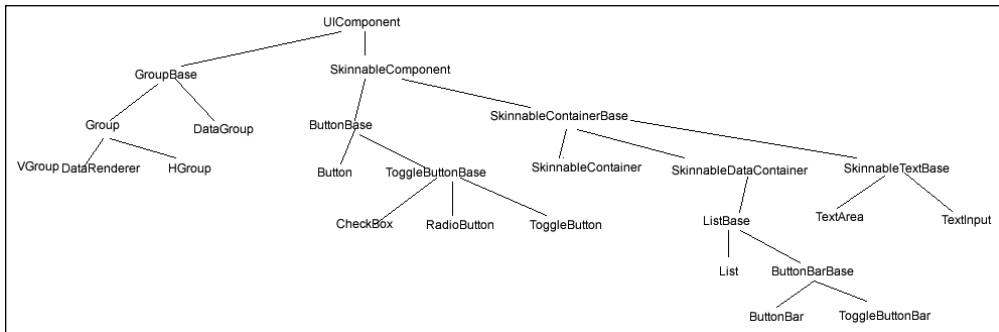
```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:BasicLayout />
  </s:layout>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
  <s:Label text="A simple component"/>
</s:Group>
```

A simple component

You will learn two things in this lesson. The first is how to build components. You'll learn the syntax and rules for creating and using custom components. Second, you will learn why you'd want to do this and how components can affect your overall application architecture. The "Introducing MXML Components" section provides an overview of how to build components. In the tasks throughout this lesson, you'll reinforce your component-building skills and continue to learn more and more details about building custom components. You'll start with a theoretical discussion of why you would want to use components. The rest of the lesson will use an architectural approach to implementing components.

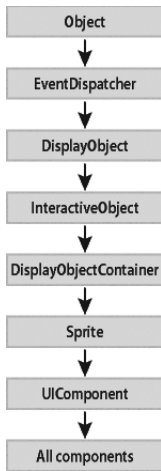
Introducing MXML Components

All Flex components and all the components you will build are ActionScript classes. The base class for the visual components you've been using and the MXML components you'll build in this lesson is `UIComponent`. In a hierarchy of components, `UIComponent` is at the top, and all the other components inherit from it.



These classes fall into general groupings based on their functionality, such as component, manager, and data service classes. In fact, `UIComponent` has itself inherited from a set of classes that provide functionality, such as event dispatching, interactivity, containment of other objects, and so on.

*** NOTE:** You can examine a complete description of the class hierarchy in the Flex ActionScript and MXML API reference, referred to as ASDoc.



Understanding the Basics of How to Create a Custom Component

When you build your own component, you basically want to do one of two things: add functionality to a predefined component, or group numerous components together.

The basic steps to build a component are as follows:

1. Create a new file with the filename you want for your component. (You don't need to actually do this; just follow along with the logic.) Because you're building a class, the name should start with an uppercase letter. Also, remember that these names will be case sensitive, like Flex in general.
2. Make the first line of code the XML document-type definition you have been using for the main application files.
3. As the first MXML tag, insert the root tag of your component, which will reflect what you want to do in the component. If it is a container, you most likely want to group several components' functionality into one easier-to-use component. If it's not a container, you most likely want to extend the functionality of a predefined component or further extend the functionality of a custom component. Either way, you will need to define in your root tag the namespaces that will be used in your component.

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


4. In the body of the component, add the functionality needed. This will vary depending on what functionality you want the component to provide.
5. In the root tag of the file that will instantiate the component, add an XML namespace so you can access the component. It's considered a best practice to group components in subdirectories according to their purpose. For instance, you will create a directory called `views`. Later in this lesson, you will add a namespace, using the word `views` as the prefix, to have access to all the custom components in the `views` directory. The statement will appear as follows:

```
xmlns:views="views.*"
```

6. Instantiate the component as you would a predefined component. For instance, if you created a file component called `UserForm.mxml`, you would instantiate that component using the namespace just created, as follows:

```
<views:UserForm/>
```

Creating a Custom Component Step by Step

Now that you know the general approach to building a component, here is a simple example of adding functionality to a predefined component. Assume that you want to build a `List` that will automatically display three grocery categories. Your component will use `<s:List>` as its root tag. Until now, all the MXML pages you've built use the `<s:Application>` tag as the root tag. Components cannot use the `<s:Application>` tag as the root tag because it can be used only once per application. Here are six steps for creating a simple component. Of course, using Flash Builder further simplifies the process by presenting a dialog to help you build a template of your component.

1. Create a file named `MyList.mxml`. (You don't need to actually do this; just follow along with the logic.)
2. The first line of the component will be the standard XML document declaration.

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

3. Because you are extending the functionality of the `<s:List>`, you will use it as the root tag. Your skeleton component will appear as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<s:List xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
</s:List>
```

4. The functionality to add to the body of the component is to display three `<s:String>` tags in the `<s:List>`. You know you use the `<s:dataProvider>` tag to supply data to an `<s:List>`, so here is the finished component:

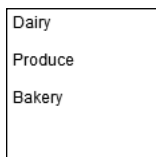
```
<?xml version="1.0" encoding="utf-8"?>
<s:List xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <s:dataProvider>
    <s:ArrayCollection>
      <fx:String>Dairy</fx:String>
      <fx:String>Produce</fx:String>
      <fx:String>Bakery</fx:String>
    </s:ArrayCollection>
  </s:dataProvider>
</s:List>
```

5. Assume that a file named `CompTest.mxml` is created at the root of the project. Also, the component is created in a directory called `myComps`. Use the word *custom* as the prefix for the components in this folder. Therefore, the XML namespace to add to the `<s:Application>` tag is `xmlns:custom="myComps.*"`.
6. Finally, instantiate the component in the main application file.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:custom="myComps.*">
  <custom:MyList/>
</s:Application>
```

- * NOTE:** You will see shortly that Flash Builder makes this process of creating the skeleton of the component even easier.

The `CompTest.mxml` output would appear as shown here.



Using Custom Components in the Application Architecture

You now know the basic mechanics of creating custom components. So now what? How does this affect what I have been doing? Why should I use them? How do I use them?

The advantages of components mentioned in the opening pages of this lesson should now be clearer:

- Components make applications easier to build, debug, and maintain.
- Components ease team development.
- With planning, components can lead to a suite of reusable code.

To facilitate using components as reusable code, you should make them independent of other code whenever possible. The components should operate as independent pieces of application logic, with a clear definition of what data must be passed into them and what data will be returned from them. The object-oriented programming term *loosely coupled* is used to describe this type of architecture.

Suppose you have a component that uses an `<s:List>` to display some information. You later learn of a new component that offers a better way to display that data. If the custom component is built correctly, you should be able to switch the display component and not need to make any other changes. You change the inner workings of the custom component, but the data going into the component and what comes out will not change, so no changes to the rest of the application are needed.

Now, you need to think about how components fit into the application architecture. Although this book is not meant to be a discourse on Flex application architectures, it would be negligent not to show how components can fit into the bigger picture. In the application you're building in this book, you will implement a primitive form of model-view-controller (MVC) architecture.

MVC is a design pattern or software architecture that separates the application's data, user interface, and control logic into three distinct groupings. The goal is to implement the logic so changes can be made to one portion of the application with minimal impact to the others. Here are some short definitions of the key terms:

- **Model:** The data the application uses. The model manages the data elements, responds to queries about its state, and manages instructions to change the data.
- **View:** The user interface. The view is responsible for presenting model data to the user and gathering information from the user.
- **Controller:** What responds to events—typically user events, but also system events. The controller interprets the events and invokes changes on the model and view.

Here is the general flow of the MVC architecture:

1. The user interacts with the user interface (a view): for example, by clicking a button to add an item to a shopping cart.
2. The controller handles the input event.
3. The controller accesses the model, maybe by retrieving or altering data, and gives the data to the view.
4. The view then uses the model data for appropriate presentation to the user.

Consider the application you are building. Eventually your FlexGrocer.mxml main application page will be part of the controller. There will be views that do the following:

- Display the different grocery item categories
- Display the items in the shopping cart
- Display a detailed view of a particular grocery item
- Display all the grocery items in a particular category

Each of these views will require some logic for interactivity. In a strict MVC architecture this code is usually located in a separate class or classes. In this book, the code will exist alongside the views as our desire is to teach Flex, not strict MVC architecture. The model is provided by the data loaded via HTTPService classes, which will soon be moved to their own classes.

Now the stage is set, and you're ready to get started building components and enhancing the architecture and functionality of the applications you are building.

Splitting Off the ShoppingView Component

This first exercise will improve the overall architecture of the application, but it will not add any functionality from the user's point of view. In fact if everything is done properly, you'll want the application to appear exactly as it did before you started. You will pull the application's visual elements into a component, a view in terms of MVC architecture. FlexGrocer.mxml will begin to transform into the controller.

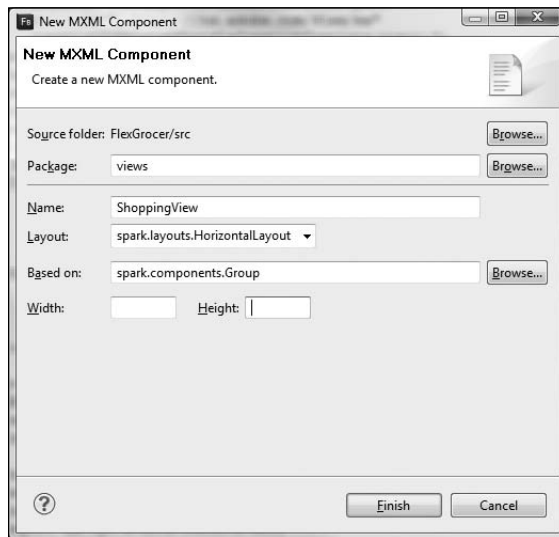
- 1 Right-click the src folder in the FlexGrocer project that you used in Lesson 8, "Using Data Bindings and Collections," and create a package named **views**.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson09/start folder.

Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

It is a best practice to organize your components. In this case, the views folder will contain the views for your applications.

- 2 Right-click the views folder and then choose New > MXML Component. In the New MXML Component dialog box, set the name as **ShoppingView**, the base component as a Group, and the layout as Horizontal. Remove any width and height values, and then click Finish.



In this case, you are using an `<s:Group>` as your root tag and are applying a `HorizontalLayout` to it, which means the children you insert in this component will be aligned horizontally beside each other.

- 3 Insert an `<fx:Script>` block just after the closing `</s:layout>` tag.
You will have an `<fx:Script>` block in this component. Some of the code you will copy from the `FlexGrocer.mxml` file, and other code you will write new.
- 4 From the `FlexGrocer.mxml` file, copy the bindable properties named `groceryInventory` and `shoppingCart` and paste them inside the `Script` block in the `ShoppingView` component. Change the scope of `groceryInventory` from a **private** variable to a **public** variable.

```
[Bindable]
public var shoppingCart:ShoppingCart = new ShoppingCart();
```

```
[Bindable]
public var groceryInventory:ArrayCollection;
```

- 5** Add the imports for the ArrayCollection and ShoppingCart classes. The easiest way to do this is to put your cursor at the end of the class name (for instance, after the final *t* of *cart*), and press Ctrl+Spacebar. You won't be prompted as to which class you want imported while importing the ArrayCollection class, but you will for the ShoppingCart class. This is because the only class that matches ArrayCollection is the ArrayCollection class, but Flash Builder will ask if you mean to use the ShoppingCart or ShoppingCartItem class, both of which start with *ShoppingCart*.

Pressing Ctrl+Spacebar will force Flash Builder to use its code-completion feature, which in this case will automatically import the classes for you.

```
<fx:Script>
  <![CDATA[
    import cart.ShoppingCart;

    import mx.collections.ArrayCollection;
    [Bindable]
    public var shoppingCart:ShoppingCart = new
      ▶ ShoppingCart();

    [Bindable]
    public var groceryInventory:ArrayCollection;

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

When you copied these variables into the component, they became properties of the component. Simply by using the var statement and defining the variables to be public, you are creating these as public properties of the components that can have data passed into them.

This is no small matter. The basic building blocks of object-oriented programming are objects, properties, and methods. So knowing how to create properties is a very important piece of information.

Later in this lesson, you will add functions to a component. Just as variables in a class are properties, so functions in the class are the methods of your components.

- 6 From the FlexGrocer application, cut the HGroup with the id of bodyGroup and all of its contents and paste it into your ShoppingView component, after the end of the <fx:Declarations> tag pair. In ShoppingView, remove the opening and closing HGroup tags, but leave the contents of this tag pair in place. You will no longer need the HGroup in place, as your ShoppingView component is set to have a HorizontalLayout.
- 7 From the FlexGrocer application, cut the VGroup set to be included in the expanded state (this VGroup holds a RichText and two Labels) and paste it into your ShoppingView component just above the VGroup with the id of cartGroup. Your resulting code should look like the following:

```

<s:VGroup id="products" width="100%" height="150"
  visible.cartView="false" width.cartView="0" height.cartView="0">
  <s:Label id="prodName" text="Milk"/>
  <s:Image scaleMode="letterbox"
    source="@Embed('assets/dairy_milk.jpg')"
    mouseOver="this.currentState='expanded'"
    mouseOut="this.currentState='State1'"/>
  <s:Label id="price" text="$1.99"/>
  <s:Button id="add" label="Add To Cart" click=
    "addToCart( groceryInventory.getItemAt( 0 ) as Product )"/>
  <s:Button id="remove" label="Remove From Cart" click=
    "removeFromCart( groceryInventory.getItemAt( 0 ) as Product )"/>
</s:VGroup>

<s:VGroup includeIn="expanded" x="200" width="100%">
  <s:RichText text="{( groceryInventory.getItemAt( 0 ) as
    Product ).description}" width="50%"/>
  <s:Label text="Certified Organic" visible="{(groceryInventory.getItemAt
    ( 0 ) as Product ).isOrganic}"/>
  <s:Label text="Low Fat" visible="{(groceryInventory.getItemAt( 0 ) as
    Product ).isLowFat}"/>
</s:VGroup>

<s:VGroup id="cartGroup" height="100%"
  width.cartView="100%">
  <s:List id="cartList"
    dataProvider="{shoppingCart.items}" includeIn="State1"/>
  <s:Label text="Your Cart Total $0"/>
  <s:Button includeIn="State1" height="17" label="View Cart"
    click="handleViewCartClick( event )"/>
  <s>DataGrid includeIn="cartView" requestedRowCount="4">
    <s:columns>
      <s:ArrayList>
        <s:GridColumn dataField="dataField1"

```

```

        headerText="Column 1"></s:GridColumn>
    <s:GridColumn dataField="dataField2"
        headerText="Column 2"></s:GridColumn>
    <s:GridColumn dataField="dataField3"
        headerText="Column 3"></s:GridColumn>
    </s:ArrayList>
</s:columns>
<s:typicalItem>
    <fx:Object dataField1="Sample Data"
        dataField2="Sample Data"
        dataField3="Sample Data"></fx:Object>
</s:typicalItem>
<s:ArrayList>
    <fx:Object dataField1="data1"
        dataField2="data1"
        dataField3="data1"></fx:Object>
    <fx:Object dataField1="data2"
        dataField2="data2"
        dataField3="data2"></fx:Object>
    <fx:Object dataField1="data3"
        dataField2="data3"
        dataField3="data3"></fx:Object>
    <fx:Object dataField1="data4"
        dataField2="data4"
        dataField3="data4"></fx:Object>
</s:ArrayList>
</s>DataGrid>
<s:Button includeIn="cartView" label="Continue Shopping"
    click="this.currentState=' '"/>
</s:VGroup>

```

- 8** Cut the states block from the main application, and paste it in the component, between the closing `<fx:Script>` tag and the starting `<fx:Declarations>` tag.

```

<s:states>
    <s:State name="State1"/>
    <s:State name="cartView"/>
    <s:State name="expanded"/>
</s:states>

```

The reality is that the order of where you place the states tag is unimportant, as long as you do not place it as the child of something other than the root node.

- 9** Cut the `addToCart()` and `removeFromCart()` methods from the main application, and paste them into your component in the `<fx:Script>` block. Use the code-completion feature to import the `Product` and `ShoppingCartItem` classes. Remove the `trace (shoppingCart);` statement from your `addToCart()` method. It is no longer needed.


```
private function addToCart( product:Product ):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( product );
    shoppingCart.addItem( sci );
}
private function removeFromCart( product:Product ):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( product );
    shoppingCart.removeItem( sci );
}
```

At this point, the full Script block should read as:

```
<fx:Script>
  <![CDATA[
    import cart.ShoppingCart;
    import cart.ShoppingCartItem;

    import mx.collections.ArrayCollection;

    import valueObjects.Product;
    [Bindable]
    public var shoppingCart:ShoppingCart = new ShoppingCart();
    [Bindable]
    public var groceryInventory:ArrayCollection;

    private function addToCart( product:Product ):void {
        var sci:ShoppingCartItem = new ShoppingCartItem( product );
        shoppingCart.addItem( sci );
    }

    private function removeFromCart( product:Product ):void {
        var sci:ShoppingCartItem = new ShoppingCartItem( product );
        shoppingCart.removeItem( sci );
    }

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

These methods are called from some of the MXML tags you moved into the `ShoppingView` class, so they need to be moved into that component.

- 10** Copy the `handleViewCartClick()` method from the main application, and paste it into your component's `<fx:Script>` block.

```
private function handleViewCartClick( event:MouseEvent ):void {
    this.currentState = "cartView";
}
```

At this point, the full Script block should read as follows:

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

```

import cart.ShoppingCart;
import cart.ShoppingCartItem;

import mx.collections.ArrayCollection;

import valueObjects.Product;
[Bindable]
public var shoppingCart:ShoppingCart = new ShoppingCart();
[Bindable]
public var groceryInventory:ArrayCollection;

private function handleViewCartClick( event:MouseEvent ):void {
    this.currentState = "cartView";
}

private function addToCart( product:Product ):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( product );
    shoppingCart.addItem( sci );
}

private function removeFromCart( product:Product ):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( product );
    shoppingCart.removeItem( sci );
}

]]>
</fx:Script>

```

These methods are called from some of the MXML tags you moved into the ShoppingView class, so they need to be moved into that component.

11 Save the file.

You have created your first MXML component. Now that the component is built, you will instantiate the new component from the main application.

12 Return to the FlexGrocer.mxml file and find the Label that shows the copyright mark. Just after that tag, type in <Shopp and choose views:ShoppingView from the code-hinting menu.



- 13** Give the new tag an id of `bodyGroup`, and make it a self-closing tag. Add a width and a height of `100%` to the tag.

```
<views:ShoppingView id="bodyGroup" width="100%" height="100%" />
```

The code-hinting feature automatically added the namespace that references the views directory to your main application tag. The Application tag should now read like this:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="handleCreationComplete( event )" xmlns:views="views.*">
```

- 14** Change the handler on the `btnCartView` button from being defined as `click`.

```
State1="handleViewCartClick( event )" to click="handleViewCartClick( event )"
```

The state definitions have all been moved to the `ShoppingView` class, so the main application no longer has any states other than the base state. For this reason, the `click` event handler should be defined without the explicit reference to `State1`.

```
<s:Button id="btnCartView" y="10" right="90" label="View Cart"
  click="handleViewCartClick( event )"/>
```

- 15** Still in `FlexGrocer.mxml`, change the `handleViewCartClick()` method so it changes the state of `bodyGroup` instead of the state of the main application.

```
private function handleViewCartClick( event:MouseEvent ):void {
  bodyGroup.currentState = "cartView";
}
```

As mentioned in the previous step, the main application no longer has states. When the user clicks the View Cart button in the control bar, you need to change the state of the `ShoppingView` in order to show the cart.

- 16** Save the main application. It should now compile with no errors. However, you have not yet passed the data needed to render the products in the shopping view, so if you run the application, it will not show any products yet. To solve this problem, in the instantiation of the `ShoppingView` component, add an attribute that binds the `groceryInventory` property from the `FlexGrocer` application into the `groceryInventory` property of the `ShoppingView` component.

```
<views:ShoppingView id="bodyGroup"
  width="100%" height="100%"
  groceryInventory="{groceryInventory}" />
```

Now the `ShoppingView` should have the data it needs to render products as it did before.

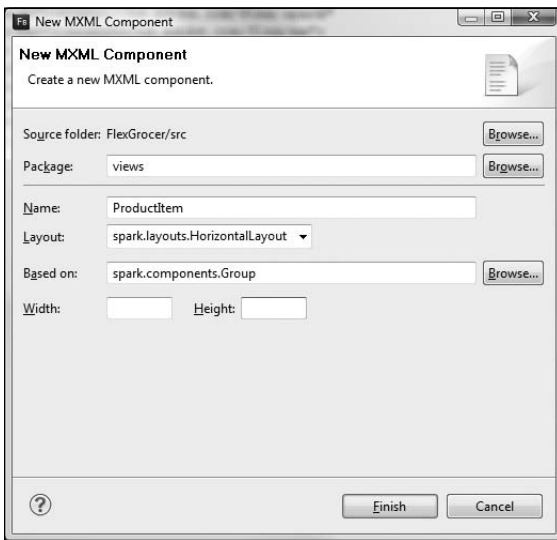
- 17** Run the `FlexGrocer.mxml` file. You see that creating the component has not changed the functionality. It still renders a product that can be added and removed from the cart.

The purpose of this first exercise was not to add functionality to the application but to refactor it. As the functionality of the application continues to grow, the main application page would have become much too long and complex. Using components gives you the chance to break it up into manageable modules.

Breaking Out a ProductItem Component

Right now, the application continues to behave as it did at the end of the previous lesson; it shows a single product and allows you to add or remove that product from the cart. However, if you wanted to show more than one item at a time, you would need to copy and paste a large block of code multiple times. Instead, you will split the elements specific to viewing a product into a separate class, and you can then create several instances of this one class to show multiple products. The new component you will create is more than just a simple view onto the data. It is intended as a reusable component that can be used anytime you need to display product information for this application. So instead of creating it in the views directory, you'll create a new components directory to hold this and other reusable components.

- 1 Right-click the src directory, and choose New Package. Name the new package **components**.
- 2 Right-click the newly created components package. Choose New > MXML Component. The name should be **ProductItem**, and the base component should be **Group** with a Horizontal layout. Remove the width and height values, and then click Finish.



- 3 Add an `<fx:Script>` tag pair just after the layout declaration.

Much as with the last component, the `Script` block will be used to add properties and methods for your new component.

- 4 Add a bindable public property named `product`, with a data type of the `Product` class, to the `Script` block. Use code-completion to import the `Product` class, or manually add an `import` statement for `valueObjects.Product`. Add another public variable, called `shoppingCart`, with a data type of `ShoppingCart`.

This will allow you to pass a specific product to each instance of this component, and pass a reference to the `shoppingCart` to each instance as well. Remember to either use the code-completion functionality when specifying the `ShoppingCart` class, or to manually add an `import` for `cart.ShoppingCart`.

```
import cart.ShoppingCart;
import valueObjects.Product;

[Bindable]
public var product:Product;

public var shoppingCart:ShoppingCart;
```

- 5 Cut the `addToCart()` and `removeFromCart()` methods from the `ShoppingView` component, and paste them into the `Script` block of the `ProductItem` component. You will need to make sure the `ShoppingCartItem` class is imported as well.

```
import cart.ShoppingCart;
import cart.ShoppingCartItem;

import valueObjects.Product;

[Bindable]
public var product:Product;

public var shoppingCart:ShoppingCart;

private function addToCart( product:Product ):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( product );
    shoppingCart.addItem( sci );
}

private function removeFromCart( product:Product ):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( product );
    shoppingCart.removeItem( sci );
}
```

- 6 In `ShoppingView.mxml`, cut the `VGroup` with the `id` of `products`, and the one that follows it, which is included in the expanded state, and paste them after the `<fx:Declarations>` section of `ProductItem`.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      import cart.ShoppingCart;
      import cart.ShoppingCartItem;

      import valueObjects.Product;
      [Bindable]
      public var product:Product;

      public var shoppingCart:ShoppingCart;

      private function addToCart(product:Product):void {
        var sci:ShoppingCartItem = new ShoppingCartItem( product );
        shoppingCart.addItem( sci );
      }

      private function removeFromCart(product:Product):void {
        var sci:ShoppingCartItem = new ShoppingCartItem( product );
        shoppingCart.removeItem( sci );
      }
    ]]>
  </fx:Script>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>

  <s:VGroup id="products" width="100%" height="150"
    visible.cartView="false" width.cartView="0" height.cartView="0">
    <s:Label id="prodName" text="Milk"/>
    <s:Image scaleMode="letterbox"
      source="@Embed('assets/dairy_milk.jpg')"
      mouseOver="this.currentState='expanded'"
      mouseOut="this.currentState='State1'"/>
    <s:Label id="price" text="$1.99"/>
    <s:Button id="add" label="Add To Cart" click="addToCart(
      groceryInventory.getItemAt( 0 ) as Product )"/>
    <s:Button id="remove" label="Remove From Cart" click=
      "removeFromCart( groceryInventory.getItemAt( 0 ) as Product )"/>
  </s:VGroup>
</s:Group>
```

```

</s:VGroup>

<s:VGroup includeIn="expanded" x="200" width="100%">
  <s:RichText text="{( groceryInventory.getItemAt( 0 ) as
    Product ).description}" width="50%" />
  <s:Label text="Certified Organic" visible=
    "{(groceryInventory.getItemAt( 0 ) as Product ).isOrganic}" />
  <s:Label text="Low Fat" visible=
    "{(groceryInventory.getItemAt( 0 ) as Product ).isLowFat}" />
</s:VGroup>
</s:Group>

```

- 7** Copy the states block from `ShoppingView.mxml`, and paste it into `ProductItem.mxml` just after the layout tag pair. In `ShoppingView`, remove the state definition for `expanded`. In `ProductItem`, remove the state definition for `cartView`.

Your `ShoppingView` states block should read like this:

```

<s:states>
  <s:State name="State1" />
  <s:State name="cartView" />
</s:states>

```

Your `ProductItem` states block should read like this:

```

<s:states>
  <s:State name="State1" />
  <s:State name="expanded" />
</s:states>

```

Now both components have a base state (`State1`), and each has another state specific for it. `ShoppingView` has the `cartView` state, which it can use to show the details of a shopping cart, and `ProductItem` has an `expanded` state, which shows expanded product details.

- 8** As `ProductItem` no longer has a `cartView` state, you need to remove the attributes of the first `VGroup` that explicitly set `width`, `height`, and `visible` properties for the `cartView` state. While removing the attributes, also remove the normal `width` and `height` attributes as well, as those will no longer be needed, either.

```

<fx:Declarations>
  <!-- Place non-visual elements (e.g., services, value objects) here -->
</fx:Declarations>
<s:VGroup id="products">
  <s:Label id="prodName" text="Milk" />
  <s:Image scaleMode="letterbox"
    source="@Embed('assets/dairy_milk.jpg')"
    mouseOver="this.currentState='expanded'"
    mouseOut="this.currentState='State1'" />...

```

- 9** Change the reference for the image source from the current embedded image "`@Embed('assets/dairy_milk.jpg')`" and instead dynamically load the image from the `assets` directory, using the image name of the product.

```
<s:Image scaleMode="letterbox"
  source="{product.imageName}"
  mouseOver="this.currentState='expanded'"
  mouseOut="this.currentState='State1'"/>
```

Your component can now show the appropriate image for any product passed to it, rather than always showing milk.

- 10** In the Label just before the image, change the `text="Milk"` to `text="{product.prodName}"` to dynamically display the product name.

```
<s:Label id="prodName" text="{product.prodName}"/>
```

Your component can now show the correct name for whichever product it has.

- 11** In the Label just after the image, change the `text="$1.99"` to `text="{product.listPrice}"` to dynamically display the product price.

```
<s:Label id="price" text="{product.listPrice}"/>
```

Your component can now show the correct price for whichever product it has.

- 12** In the click handlers for both the Add To Cart and Remove From Cart buttons, change the argument passed to the function from `groceryInventory.getItemAt(0)` as `Product` to `product`.

```
<s:Button label="Add To Cart" id="add"
  click="addToCart( product )"/>
<s:Button label="Remove From Cart" id="remove"
  click="removeFromCart( product )"/>
```

Since your component is no longer dealing with the entire `groceryInventory` collection, but instead with an individual product, the reference to the product for this component is now greatly simplified. When you create an instance of this component from the `ShoppingView` component, you will pass just one specific product to each instance.

- 13** For the RichText control and two labels in the `VGroup` shown in the expanded view, change the reference in the binding from `groceryInventory.getItemAt(0)` as `Product` to just `product`.

```
<s:VGroup includeIn="expanded" x="200" width="100%">
  <s:RichText text="{product.description}" width="50%"/>
  <s:Label text="Certified Organic" visible="{product.isOrganic}"/>
  <s:Label text="Low Fat" visible="{product.isLowFat}"/>
</s:VGroup>
```


Your final code for the ProductItem component should read like this:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>

  <s:states>
    <s:State name="State1"/>
    <s:State name="expanded"/>
  </s:states>

  <fx:Script>
    <![CDATA[
      import cart.ShoppingCart;
      import cart.ShoppingCartItem;

      import valueObjects.Product;
      [Bindable]
      public var product:Product;

      public var shoppingCart:ShoppingCart;

      private function addToCart(product:Product):void {
        var sci:ShoppingCartItem = new ShoppingCartItem( product );
        shoppingCart.addItem( sci );
      }

      private function removeFromCart(product:Product):void {
        var sci:ShoppingCartItem = new ShoppingCartItem( product );
        shoppingCart.removeItem( sci );
      }
    ]]>
  </fx:Script>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>

  <s:VGroup id="products">
    <s:Label id="prodName" text="{product.prodName}"/>
    <s:Image scaleMode="letterbox"
      source="assets/{product.imageName}"
      mouseOver="this.currentState='expanded'"
      mouseOut="this.currentState='State1'"/>
  </s:VGroup>
</s:Group>
```

```

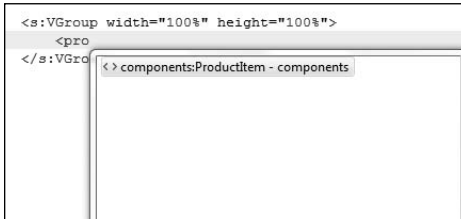
    <s:Label id="price" text="{product.listPrice}"/>
    <s:Button id="add" label="Add To Cart" click="addToCart( product )"/>
    <s:Button id="remove" label="Remove From Cart" click=
        "removeFromCart( product )"/>
</s:VGroup>

<s:VGroup includeIn="expanded" x="200" width="100%">
    <s:RichText text="{product.description}" width="50%"/>
    <s:Label text="Certified Organic" visible="{product.isOrganic}"/>
    <s:Label text="Low Fat" visible="{product.isLowFat}"/>
</s:VGroup>
</s:Group>

```

Next, you will need to create one or more instances of this component from the `ShoppingView`.

- Switch back to `ShoppingView.mxml`. After the `<fx:Declaration>` tag pair, but before the `cartGroup VGroup`, create a new `VGroup` with a width and a height of `100%`. Inside this group, create an instance of `ProductItem`. If you begin typing it and use code-completion, as you did when you created the instance of `ShoppingView` in the previous lesson, the import statement for the components package will be automatically added.



- Give the new `ProductItem` instance an `id="product1"`, specify a width and a height of `100%`, bind a reference of the local `shoppingCart` into the `shoppingCart` property of the new component, and bind `groceryInventory.getItemAt(0)` as `Product` to its `product` property.

```

<s:VGroup width="100%" height="100%">
    <components:ProductItem id="product1"
        width="100%" height="100%"
        shoppingCart="{shoppingCart}"
        product="{groceryInventory.getItemAt(0) as Product}"/>
</s:VGroup>

```

16 Save all the files and run the application.



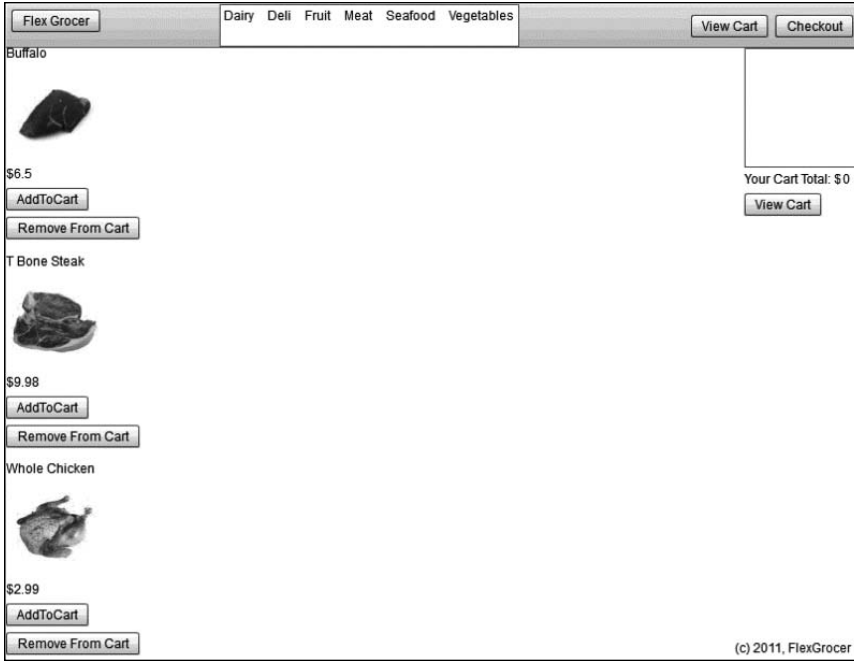
Your application is now displaying the first item from the groceryInventory, which is buffalo meat, rather than the milk you have been using in the application thus far. But wait, there's more. Since you now have a component that can easily show individual products, you can show several at once.

17 Switch back to ShoppingView.mxml. Copy the tag that creates an instance of ProductItem, and paste it twice more, just below the current one. Change the id of the new ones to be product2 and product3. Also change the binding to the product property to use item 1 and item 2, while the original is getting item 0.

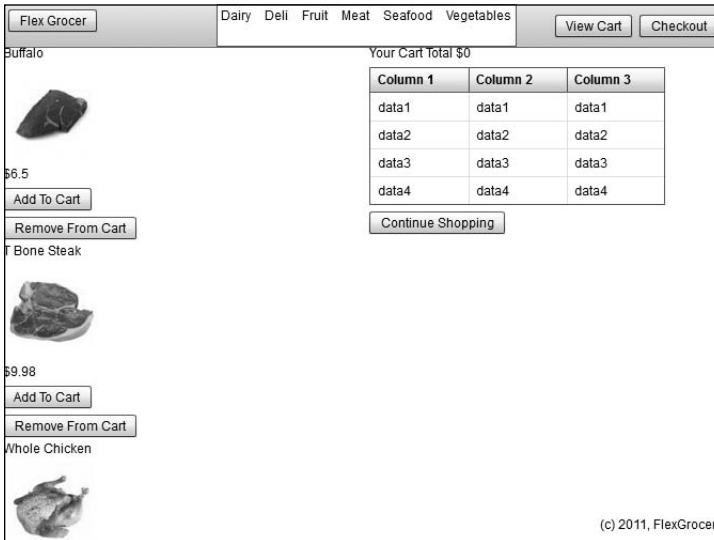
```
<s:VGroup width="100%" height="100%">
  <components:ProductItem id="product1"
    width="100%" height="100%"
    shoppingCart="{shoppingCart}"
    product="{groceryInventory.getItemAt(0) as Product}"/>
  <components:ProductItem id="product2"
    width="100%" height="100%"
    shoppingCart="{shoppingCart}"
    product="{groceryInventory.getItemAt(1) as Product}"/>
  <components:ProductItem id="product3"
    width="100%" height="100%"
    shoppingCart="{shoppingCart}"
    product="{groceryInventory.getItemAt(2) as Product}"/>
</s:VGroup>
```

Now, as you save and run the application, you should see several products shown.

*** NOTE:** In the next lesson, you will learn how to use a DataGroup to create one ProductItem for each Product in the groceryInventory collection.



One bug still needs to be fixed. If you click the View Cart button, the products are still being shown, rather than being hidden. You will fix that in the next step.



- 18 In the VGroup that contains the ProductItems, specify values for the width and the height of the cartView state to be 0, and the visible property to be false.

```
<s:VGroup width="100%" height="100%"
    width.cartView="0" height.cartView="0"
    visible.cartView="false">
```

The application should now be able to switch between the various states of the ProductItem and ShoppingView components correctly, while now displaying three products instead of just one.

Creating Components to Manage Loading the Data

In the first exercise, you refactored part of the application without adding any functionality. In the second exercise, you added functionality (showing multiple products) while building another component. This exercise is akin to the first, in which you are refactoring the application without adding any visible functionality for the user.

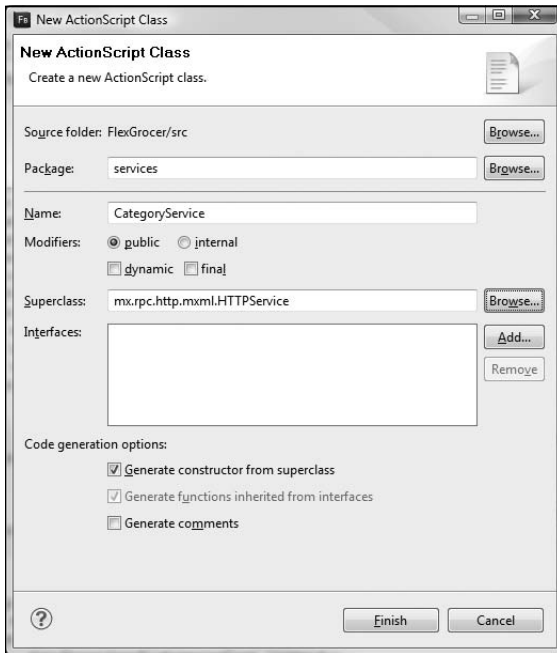
Right now, the main application file is a bit cluttered by the instantiation and event handlers for the two HTTPServices. In this exercise, you're going to create ActionScript classes for these services, which will contain the HTTPService components as well as result and fault event handlers and will expose the data loaded through public bindable properties. The new class will provide certain types of data to all the applications when they need it. This data manager component will be different from other components you've built in this lesson in that it will not have any representation that a user will see. Such a component is referred to as a non-visual component.

- 1 Create a new services package in the FlexGrocer project.

Alternatively, if you didn't complete the previous exercise or your code is not functioning properly, you can import the FlexGrocer-PreData.fxp project from the Lesson09/intermediate folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

Because the new components are neither a value object nor a view, a new package is needed to continue organizing your components by function.

- 2 Right-click the services folder and then choose New ActionScript Class. In the New ActionScript Class dialog box, set the name as **CategoryService** and set the superclass as a **mx.rpc.http.mxml.HTTPService**; leave the rest of the defaults, then click Finish.



As you want a class that provides all the functionality of `HTTPService`, but has some additional methods and properties, `HTTPService` is the most logical choice for a base class.

- 3 After the line declaring `public class CategoryService` but before the constructor, add a bindable public variable `categories:XMLListCollection`.

```
import mx.collections.XMLListCollection;
import mx.rpc.http.mxml.HTTPService;

public class CategoryService extends HTTPService {
    [Bindable]
    public var categories:XMLListCollection;

    public function CategoryService(rootURL:String=null,
destination:String=null)
```

This `categories` property will determine how other classes interact with the data loaded by the service. Don't forget to use the code-hinting feature, or to manually import the `XMLListCollection` class.

- 4** In the constructor, after the call to `super()`, set the `resultFormat` property of your class equal to `e4x` and the `url` property to `http://www.flexgrocer.com/category.xml`.

```
public function CategoryService(rootURL:String=null, destination:String=null)
{
    super(rootURL, destination);
    this.resultFormat="e4x";
    this.url="http://www.flexgrocer.com/category.xml";
}
```

Take a look at the constructor here. The first line inside the function definition (which was automatically added by the new-class wizard), passes the `rootURL` and `destination` arguments to the constructor of the superclass. This way, it is not necessary to duplicate the logic found in the superclass's constructor. The two lines you added are setting the `resultFormat` and `url` properties of the `HTTPService` class, as you learned in previous lessons.

- 5** Open `FlexGrocer.mxml`, cut the `handleCategoryResult()` method, and paste it into the new `CategoryService` class, after the constructor.

```
private function handleCategoryResult( event:ResultEvent ):void {
    categories = new XMLListCollection( event.result.category );
}
```

As with each new class you introduce, make sure the other classes you are using get imported. In `CategoryService`, you need to ensure that the `ResultEvent` class gets imported, either by typing in the `import` statement for `mx.rpc.events.ResultEvent` manually, or by using the code-completion feature. This method will populate the `categories` property with the results from the service call.

- 6** In the constructor, add an event listener for the `result` event. Set `handleCategoryResult` as the handler for that event.

```
addEventListener(ResultEvent.RESULT, handleCategoryResult);
```

The `addEventListener()` method allows you to specify an event to listen for (in this case it's the event result and a method that will be used as the event handler).

- 7** Save `CategoryService`.

Your service class is now complete. All that remains is to use it in the application. The completed `CategoryService` class should read like this:

```
package services {
    import mx.collections.XMLListCollection;
    import mx.rpc.events.ResultEvent;
    import mx.rpc.http.mxml.HTTPService;

    public class CategoryService extends HTTPService {
```

```

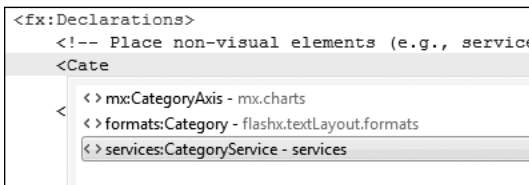
[Bindable]
public var categories:XMLListCollection;

public function CategoryService(rootURL:String=null, destination:
    String=null)
{
    super(rootURL, destination);
    this.resultFormat = "e4x";
    this.url = "http://www.flexgrocer.com/category.xml";
    addEventListener(ResultEvent.RESULT, handleCategoryResult);
}

private function handleCategoryResult(event:ResultEvent):void {
    categories = new XMLListCollection( event.result.category );
}
}
}
}

```

- 8** Switch to the FlexGrocer.mxml file. In the `<fx:Declarations>` block of FlexGrocer.mxml, delete the `<s:HTTPService>` tag with the id of `categoryService`.



- 9** In its place, create an instance of the `CategoryService` class. Give this new instance an id of `categoryService`.

As with the previous components you instantiate, if you use the code-hinting features, the namespace will be automatically added for you.

```
<services:CategoryService id="categoryService"/>
```

You now have your new component being used in place of (and in fact with the same id as) the previous `HTTPService`. Since the id is the same, the existing call to `categoryService.send()` in the `handleCreationComplete()` method does not need to change.

- 10** Remove the bindable private `categories` property.

You will no longer need this, as the categories are now available from the `categoryService` instance directly.

- 11** Find the List class created inside the controlBarContent. Change the dataProvider from categories to categoryService.categories.

```
<s:List left="200" height="40" dataProvider="{categoryService.categories}"
  labelField="name">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
</s:List>
```

The FlexGrocer application is now using your new CategoryService class, instead of having the service properties and handlers all coded into the main application.

- 12** Save all your files and run the FlexGrocer application. It should now behave as it always did, with the categories loaded into the horizontal list.



Next, you will create a service class similar to CategoryService to load and manage the products, and you'll remove that logic from the main application.

- 13** Close all your open files. Right-click the services folder and then choose New ActionScript Class. In the New ActionScript Class dialog box, set the Name as **ProductService** and set the Superclass as `mx.rpc.http.mxml.HTTPService`; leave the rest of the defaults, then click Finish.
- 14** After the line declaring `public class ProductService` but before the constructor, add a bindable public variable `products:ArrayCollection`.

```
[Bindable]
public var products:ArrayCollection;
```

This `products` property will determine how other classes interact with the data loaded by the service. Don't forget to use the code-completion feature, or to manually import the `ArrayCollection` class.

- 15** In the constructor, after the call to `super()`, set the `resultFormat` property of your class equal to `e4x` and the `url` property to `http://www.flexgrocer.com/categorizedProducts.xml`.

```
public function ProductService(rootURL:String=null, destination:String=null)
{
    super(rootURL, destination);
    this.resultFormat="e4x";
    this.url="http://www.flexgrocer.com/categorizedProducts.xml";
}
```

The constructor is just like the one for the `CategoryService`, with a different `url` property.

- 16** Open `FlexGrocer.mxml`, cut the `handleProductResult()` method, and paste it into the new `ProductService` class, after the constructor. Change the final line so that it populates the `products` property rather than the `groceryInventory` property. Change the local `products` variable to be `productsArray`.

```
private function handleProductResult( event:ResultEvent ):void {
    var productsArray:Array = new Array();
    var resultData:XMLList = event.result..product;

    for each (var p:XML in resultData) {
        var product:Product = Product.buildProductFromAttributes( p );
        productsArray.push( product );
    }

    products = new ArrayCollection( productsArray );
}
```

As with each new class you introduce, make sure you import the newly referenced classes (`ResultEvent` and `Product`), either by typing in the import, or by using the code-completion feature. This method will parse the results of the service call into `Product` instances and populate the `products` property with them.

- 17** In the constructor, add an event listener for the `result` event. Set `handleProductResult()` method as the handler for that event.

```
addEventListener(ResultEvent.RESULT, handleProductResult);
```

Just as with the `CategoryService` class, you will want to listen for the `result` event, and pass the results on to a handler method. The final `ProductService` class should read like this:

```
package services {
    import mx.collections.ArrayCollection;
    import mx.rpc.events.ResultEvent;
    import mx.rpc.http.mxml.HTTPService;

    import valueObjects.Product;

    public class ProductService extends HTTPService {
```

```

[Bindable]
public var products:ArrayCollection;
public function ProductService(rootURL:String=null,
    ↳ destination:String=null){
    super(rootURL, destination);
    addEventListener(ResultEvent.RESULT,
        ↳ handleProductResult);
    this.resultFormat="e4x";
this.url="http://www.flexgrocer.com/categorizedProducts.xml";
}
private function handleProductResult(
    ↳ event:ResultEvent ):void {
    var productsArray:Array = new Array();
    var resultData:XMLList =
        ↳ event.result..product;

    for each (var p:XML in resultData) {
        var product:Product =
            ↳ Product.buildProductFromAttributes( p );
        productsArray.push( product );
    }

    products = new ArrayCollection( productArray );
}
}
}
}

```

18 Save ProductService. Switch to the FlexGrocer.mxml file.

Your service class is now complete. All that remains is to use it in the application.

In the `<fx:Declarations>` block of FlexGrocer.mxml, delete the `<s:HTTPService>` tag with the id of `productService`. In its place, create an instance of the `ProductService` class. Give this new instance an id of `productService`.

As with the previous components you instantiate, if you use the code-hinting feature, the namespace will be automatically added for you.

```
<services:ProductService id="productService"/>
```

Since the id is the same, the existing call to `productService.send()` in the `handleCreationComplete()` method does not need to change.

19 Remove the bindable private `groceryInventory` property and the Bindable public `shoppingCart` property.

You will no longer need these, as the products are now available from the `productService` instance's `products` property and the `ShoppingCart` is now defined in the `ShoppingView`.

- 20** With the exception of `mx.events.FlexEvent` you can now remove all the imports from this file. They are no longer needed as the functionality has been moved to components.
- 21** Find the instance of the `ShoppingView` class within `FlexGrocer.mxml`. Change `groceryInventory="{groceryInventory}"` to `groceryInventory="{productService.products}"`.

```
<views:ShoppingView id="bodyGroup"
    width="100%" height="100%"
    groceryInventory="{productService.products}" />
```

Your refactoring of the `FlexGrocer` application into components is now complete.

- 22** Save all your files and run the `FlexGrocer` application. It should now behave as it always did, but now in an easier-to-maintain fashion.

Your refactored `FlexGrocer` file should now read like this:

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

    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
        <services:CategoryService id="categoryService"/>

        <services:ProductService id="productService"/>
    </fx:Declarations>

    <fx:Script>
        <![CDATA[
            import mx.events.FlexEvent;

            private function handleViewCartClick( event:MouseEvent ):void {
                bodyGroup.currentState="cartView";
            }

            private function handleCreationComplete(event:FlexEvent):void {
                categoryService.send();
                productService.send();
            }

        ]]>
    </fx:Script>

    <s:controlBarLayout>
        <s:BasicLayout />
```

```
</s:controlBarLayout>

<s:controlBarContent>
  <s:Button label="Flex Grocer" x="5" y="5" />
  <s:Button id="btnCartView" y="10" right="90" label="View Cart"
    click="handleViewCartClick( event )"/>
  <s:Button id="btnCheckout" y="10" right="10" label="Checkout"/>
  <s:List left="200" height="40" dataProvider=
    "{categoryService.categories}" labelField="name">
    <s:layout>
      <s:HorizontalLayout/>
    </s:layout>
  </s:List>
</s:controlBarContent>

<s:Label right="10" bottom="10" text="(c) 2011, FlexGrocer"/>

<views:ShoppingView id="bodyGroup"
  width="100%" height="100%"
  groceryInventory="{productService.products}"/>
</s:Application>
```

What You Have Learned

In this lesson, you have:

- Gained a theoretical understanding of why components should be used and how they fit into a simple implementation of MVC architecture (pages 208–213)
- Built a component that moved the visual elements from a main application page to the component and then instantiated the component in the main application page (pages 213–230)
- Created non-visual components that provide category and product information to the applications (pages 230–238)

This page intentionally left blank

LESSON 10

What You Will Learn

In this lesson, you will:

- Populate a List control with a dataset
- Populate a DataGroup with a dataset and display the information using a renderer
- Create an MXML component to be used as a renderer
- Use the Generate Getter/Setter wizard
- Learn about virtualization
- Respond to a user's selection from a list

Approximate Time

This lesson takes approximately 2 hours to complete.

LESSON 10

Using DataGroups and Lists

In this lesson, you'll develop your skill in working with datasets. A dataset is really nothing but several data elements consolidated in a single object, like an Array, XMLList, ArrayCollection, or XMLListCollection. Up to this point, you've learned a few ways to display, manipulate, or loop over these datasets. In this chapter, you'll learn about Flex components that automatically create a visual element for each item in a dataset.



*Datasets used with horizontally arranged List to display categories
and with a DataGroup to display grocery items*

In this lesson, you'll learn about Lists and DataGroups. Both List and DataGroup instances can create a visual element for each item in its dataset (which is set to the DataGroup's `dataProvider` property). What is shown for each element will depend on the `itemRenderer` being used. You'll learn about `itemRenderers` in this lesson as well.

The List class, much like the DataGroup class, has a dataset in its `dataProvider` and will visually represent each item using its `itemRenderer`. Lists add another piece of functionality, in that they manage the user's selection of items from the list and provide an API for determining which item(s) if any, are selected.

In the course of this lesson, you'll rework the `ShoppingView` component. Instead of having a hard-coded set of `ProductItems` as children, the component uses a `DataGroup` to dynamically create one `ProductItem` for each element in the `groceryInventory` `ArrayCollection`. In this process, you'll rework the `ProductItem` class to be an `itemRenderer`. You'll also finish building out the functionality of the List displaying categories at the top of the application and will learn how to make the `ShoppingView` change the contents of its `groceryInventory` property when the user selects one of the categories.

Using Lists

In the application, you have already used two List instances, one with a horizontal layout to display the categories across the top of the application, and the other to display the items in the shopping cart. From your use of these two Lists, you know that the List class is provided with a dataset via `dataProvider` property (one list is using a `XMLListCollection`, and the other an `ArrayCollection`), and the list will display one item for each element in its `dataProvider`.

In Lesson 6, "Using Remote XML Data," you used a list to display the categories in the control bar. In that list, you specified a `labelField` to indicate which property the list should display. Using the `labelField` property is a very effective way of specifying which property of an object will be shown for each item of the list; however, it is limited in that it can display only text. If you want to format the data, or concatenate multiple properties, you'll need to use a `labelFunction`.

Using a labelFunction with a List

A `labelFunction` is a function that is used to determine the text to be rendered for each item in a List. This is done with the `labelFunction` property. The function will accept an `Object` as a parameter (if you are using strongly typed objects, you can specify the actual data type instead of the generic). This parameter represents the data to be shown for each item displayed by the List. The following code shows an example of a `labelFunction`, which displays the category of an item with its name and cost.

```

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

  <fx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      private var dp:ArrayCollection;

      private function generateCollection():void{
        var arrayData:Array = new Array();
        var o1:Object = new Object();
        o1.name = "banana";
        o1.category="fruit";
        o1.cost=.99;
        arrayData.push(o1);
        var o2:Object = new Object();
        o2.name = "bread";
        o2.category="bakery";
        o2.cost=1.99;
        arrayData.push(o2);
        var o3:Object = new Object();
        o3.name = "orange";
        o3.category="fruit";
        o3.cost=.52;
        arrayData.push(o3);
        var o4:Object = new Object();
        o4.name = "donut";
        o4.category="bakery";
        o4.cost=.33;
        arrayData.push(o4);
        var o5:Object = new Object();
        o5.name = "apple";
        o5.category="fruit";
        o5.cost=1.05;
        arrayData.push(o5);
        dp = new ArrayCollection(arrayData);
      }

      private function multiDisplay(item:Object):String{
        return item.category+": "+item.name+" $" +item.cost;
      }
    ]]>
  </fx:Script>

  <s>List dataProvider="{dp}"
    labelFunction="multiDisplay"
  />

</s:Application>

```

If you saved and ran this application, it would appear like this:

```
fruit: banana $0.99
bakery: bread $1.99
fruit: orange $0.52
bakery: donut $0.33
fruit: apple $1.05
```

Each object from the `dp ArrayCollection` is passed into the `labelFunction()` before it is rendered, and whatever value is returned from that function is what will be shown. In this case, you are displaying the category name, the item's name, and then its cost.

*** NOTE:** Although the `multiDisplay` function accepts parameters

```
private function multiDisplay(item:Object):String
```

you only pass a reference to the function to the List's `labelFunction` property.

```
labelFunction="multiDisplay"
```

Flex will automatically call the function with the correct arguments as it renders each item from the `dataProvider`.

In this next exercise, you'll use a `labelFunction` to format the data rendered in the shopping cart list.

1 Open the `ShoppingView` class.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer.fxp` project from the `Lesson10/start` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

2 Create a private function named `renderProductName()`, which accepts a `ShoppingCartItem` as a parameter and returns a `String`.

```
private function renderProductName( item:ShoppingCartItem ):String {
}
}
```

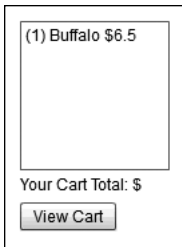
3 As the first line of the function, create a local variable, `data` typed as a `Product`, which is equal to the `product` property of the parameter to the function. Then, construct and return a string that concatenates parentheses around the `item.quantity`, followed by `product.prodName`, a dollar sign, and then the item's subtotal.

```
private function renderProductName( item:ShoppingCartItem ):String {
    var product:Product = item.product;
    return '(' + item.quantity + ')' + product.prodName + ' $' + item.subtotal;
}
```

- 4 Find the list in the `cartGroup`, and instruct it to use the `renderProductName` `labelFunction`.

```
<s:List id="cartList"
  dataProvider="{shoppingCart.items}"
  includeIn="State1"
  labelFunction="renderProductName"/>
```

- 5 Save and run the application. Notice how the items are formatted in the cart as you add products.



Using DataGroups

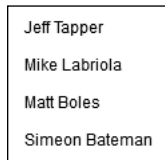
In previous lessons, you learned that the Flex 4.x framework includes a container class named `Group`, which can be used to contain any arbitrary visual elements as children and to which a layout can be applied. A `DataGroup` follows the same concept, but rather than requiring the number of children to be explicitly defined, it allows you to pass a dataset, and it will automatically create one visual child for each item in the dataset. Take a look at this simple example:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:DataGroup itemRenderer="spark.skins.spark.DefaultItemRenderer">
    <s:dataProvider>
      <s:ArrayList>
        <fx:String>Jeff Tapper</fx:String>
        <fx:String>Mike Labriola</fx:String>
        <fx:String>Matt Boles</fx:String>
        <fx:String>Steve Lund</fx:String>
      </s:ArrayList>
    </s:dataProvider>
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
</s:DataGroup>
</s:Application>
```

Here, you have a simple Flex application with only one child, a `DataGroup` container. The `DataGroup` is instructed to use a class called `DefaultItemRenderer` to render each item. You'll examine the `DefaultItemRenderer` and alternatives to it shortly.

After the items are rendered, a dataset is assigned to the `DataGroup`'s `dataProvider` property. In this case, the dataset is an `ArrayList`. In Lesson 8, "Using Data Binding and Collections," you learned that `ArrayCollections` not only provide the benefit of data binding but also have a rich set of additional features for sorting, filtering, and finding data quickly. An `ArrayList` is like an `ArrayCollection` in that it proxies an `Array` to provide data binding. Unlike the `ArrayCollection`, the `ArrayList` does not provide the additional functionality of sorting, filtering, or searching for items. This is why the `ArrayList` can be thought of as a lighter-weight version of the `ArrayCollection` class, concerned only with providing bindability to an underlying `Array`.

The `DataGroup` has its layout set to be vertical. When this runs, four instances of the `DefaultItemRenderer` will be created, one for each item in the `ArrayList`. The renderer will use a `Label` component to show each item.



Implementing an `ItemRenderer`

As you saw in the previous example, you tell the `DataGroup` how to display the elements from its `dataProvider` by specifying a class to be used as its `itemRenderer`. In the last example, the `DefaultItemRenderer` class was utilized, which simply uses a label to display each element. You can easily create your own `itemRenderer` as well.

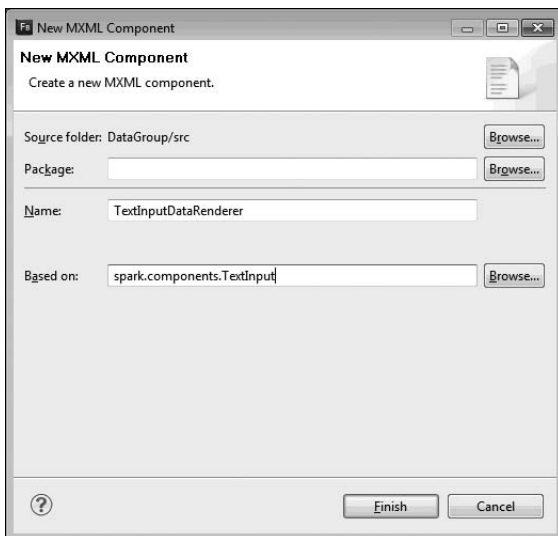
When you create your own `itemRenderers`, your new class can either implement the `IDataRenderer` interface, or you can subclass a class that already implements that interface, such as the `DataRenderer` class. The `IDataRenderer` interface simply dictates that the implementing classes have get and set functions for the `data` property, which is data-typed generically as an `Object`. The way the `itemRenderer` generally works is that one instance of the renderer will be created for each element in the `dataProvider` (this isn't entirely true, but the nuances of this function will be revealed later in this lesson, when you learn about virtualization), and the `data` property of the `itemRenderer` will be set with the data for that element in the `dataProvider`.

In this exercise, you'll create an itemRenderer that implements the IDataRenderer interface and displays the element in a TextInput instead of a Label.

- 1 Import the DataGroup.fxp from the Lesson10/independent directory into Flash Builder. Please refer to the appendix for complete instructions on importing a project.

In the DataGroup.mxml file in the default package of the src directory, you'll find the code base shown in the previous section.

- 2 Right-click the src folder of the DataGroup project, and choose New MXML Component. Leave the package blank. Specify the name as **TextInputDataRenderer**, and set it to be based on **spark.components.TextInput**. Click Finish.



This will create an MXML file with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<s:TextInput xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
</s:TextInput>
```

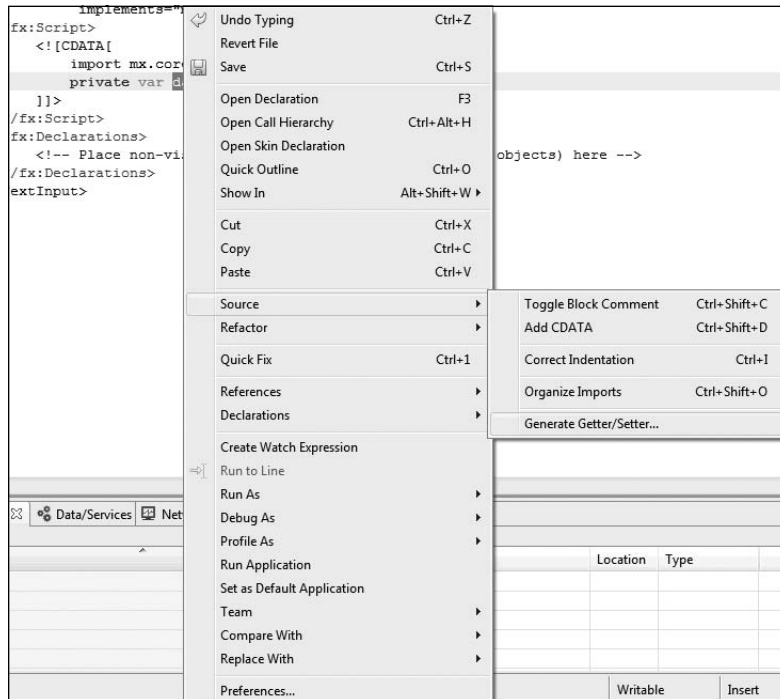
- 3** Add an attribute to the `<s:TextInput tag>`, setting an `implements` attribute equal to the value `mx.core.IDataRenderer`.

```
<?xml version="1.0" encoding="utf-8"?>
<s:TextInput xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  implements="mx.core.IDataRenderer">
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
</s:TextInput>
```

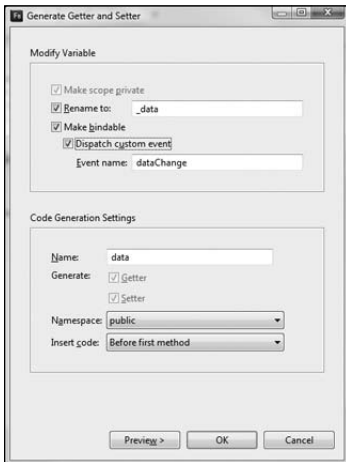
- 4** Add a new `<fx:Script>` block and a private variable, called `data`, with a data type of `Object`.

```
<fx:Script>
  <![CDATA[
    private var data:Object;
  ]]>
</fx:Script>
```

- 5** Select the data element, right-click it, and choose `Source > Generate Getter/Setter`.



- 6 In the Generate Getter and Setter dialog box, select the Make bindable check box and the Dispatch custom event check box. When your dialog box looks like the following image, click OK.



This wizard will create the public get and set functions for the `data` property and rename the private `data` to `_data`. The resulting code will look like this:

```
private var _data:Object;

[Bindable(event="dataChange")]
public function get data():Object
{
    return _data;
}

public function set data(value:Object):void
{
    if( _data !== value)
    {
        _data = value;
        dispatchEvent(new Event("dataChange"));
    }
}
```

As you learned in Lesson 8, code constructed in this way indicates that any elements bound to this class's `data` property will be updated automatically when this class dispatches an event named `dataChange`.

- 7** In the root tag, bind the text property to the toString() method of the data property.

Your renderer is now complete. All that remains is to tell the DataGroup to use it.

The complete code for the renderer should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<s:TextInput xmlns:fx="http://ns.adobe.com/mxml/2009"
             xmlns:s="library://ns.adobe.com/flex/spark"
             implements="mx.core.IDataRenderer"
             text="{data.toString()}">
  <fx:Script>
    <![CDATA[
      private var _data:Object;

      [Bindable(event="dataChange")]
      public function get data():Object
      {
        return _data;
      }

      public function set data(value:Object):void
      {
        if( _data !== value)
        {
          _data = value;
          dispatchEvent(new Event("dataChange"));
        }
      }

    ]]>
  </fx:Script>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
</s:TextInput>
```

- 8** Switch back to DataGroup.mxml. Change the itemRenderer of the DataGroup to use your newly created TextInputDataRenderer instead.

```
<s:DataGroup itemRenderer="TextInputDataRenderer">
```

- 9** Save and run the application. Notice that this time, the elements are rendered as TextInputs, rather than as Labels.

Jeff Tapper
Mike Labriola
Matt Boles
Simeon Bateman

An alternative to implementing the `IDataRenderer` class yourself is to use a base class, such as the `DataRenderer` class, that already implements this class. You'll do this in the next exercise as you change `ProductItem` to be a `DataRenderer`.

▼ **IMPORTANT!** For the remainder of this lesson, you'll not be able to add or remove products from your shopping cart. That is a consequence of the major refactor you are about to perform. However, you'll make it work again in the next lesson.

Using a DataGroup in the ShoppingView

In this exercise, you'll switch the `VGroup` that has the `ProductItem` instances to be a `DataGroup` that uses `ProductItem` as a `DataRenderer`.

- 1 Open the `ProductItem.mxml` from the `FlexGrocer` project file that you used earlier in this lesson.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer-PreDataRenderer.fxp` project from the `Lesson10/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 In `ProductItem.mxml`, change the opening and closing tags from `Group` to `DataRenderer`. Add a `width="100%"` attribute to the tag.

As mentioned earlier, the `DataRenderer` class is a subclass of `Group` that implements the `IDataRenderer` interface.

```
<s:DataRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  width="100%">
  ...
</s:DataRenderer>
```

- 3 In the `Script` block, override the data setter and set the class's `product` property to the value passed to the function. You'll need to cast the value as a `Product`.

```
override public function set data(value:Object):void{
    this.product = value as Product;
}
```

Overriding means that you are changing the behavior of the `set data` method in the `DataRenderer` class and replacing it with your own behavior in this class. If you are unfamiliar with the concept of overriding a method, please refer to the many great articles on object oriented programming on Wikipedia.

With this small change, your `ProductItem` class can now function as a `DataRenderer`. Each time the `data` property is set, it is in turn passed to the `product` property, which is already bound to the controls. Next you'll change the `ShoppingView` class to use a `DataGroup` with your new renderer.

- 4 Open `ShoppingView.mxml`. Find the `VGroup` that contains the three `ProductItem` instances. Change the opening and closing `VGroup` tags to be `DataGroup` tags instead. Remove the three `ProductItem` instances that are the children.

```
<s:DataGroup width="100%" height="100%"
    width.cartView="0" height.cartView="0"
    visible.cartView="false">
</s:DataGroup>
```

Next, you'll need to specify the `dataProvider` and `itemRenderer`.

- 5 Add an `itemRenderer` attribute to the opening `DataGroup` tag, which specifies `components.ProductItem` as the `itemRenderer`.

```
<s:DataGroup width="100%" height="100%"
    width.cartView="0" height.cartView="0"
    visible.cartView="false"
    itemRenderer="components.ProductItem">
</s:DataGroup>
```

- 6 Add a `dataProvider` attribute to the `DataGroup`, which is bound to the `groceryInventory` property.

```
<s:DataGroup x="0" y="0" width="100%" height="100%"
    width.cartView="0" height.cartView="0"
    visible.cartView="false"
    itemRenderer="components.ProductItem"
    dataProvider="{groceryInventory}">
</s:DataGroup>
```

If you save the files and run the application, you'll see the products are all rendered on top of each other, with the text being unreadable. This is happening because you haven't specified a layout object for the `DataGroup` to use.



7 As a child tag to the `DataGroup`, specify a `VerticalLayout` instance as the value of the `layout` property.

```
<s:DataGroup x="0" y="0" width="100%" height="100%"
  width.cartView="0" height.cartView="0"
  visible.cartView="false"
  itemRenderer="components.ProductItem"
  dataProvider="{groceryInventory}">
  <s:layout><s:VerticalLayout/></s:layout>
</s:DataGroup>
```

Now as you save and run the application, the products render properly.



*** NOTE:** The “add to cart” functionality no longer works when you have completed these steps. This is expected. In the next lesson, you’ll use events to fix this problem.

Understanding Virtualization

Each visual object takes processor time to create and RAM to store. It is inherently inefficient to create and store visual objects that are not displayed to the user. Virtualization solves this problem by creating visual objects only for the elements that will be seen. In situations in which the user needs to scroll to see more elements, the objects are not created initially. Instead, as the user scrolls, the objects that are scrolled off the screen are recycled and reset to display the new elements that are being scrolled on-screen.

With virtualization, if a dataset of 1000 items is set in a `DataGroup` that has room to show 10 renderers, the application will need to create only 10 instances of the renderers rather than 1000, greatly reducing the impact on the processor and RAM.

To enable virtualization for a `DataGroup`, you set the `useVirtualLayout` property of the `Layout` class to `true` (it is `false` by default).

```
<s:layout>
  <s:VerticalLayout useVirtualLayout="true"/>
</s:layout>
```

As you know, the layout objects are used by many Flex components, not just `DataGroups`. However, not all these support virtualization. If you try to specify a layout to use virtualization in a component that does not support virtualization, the component will simply ignore that attribute of the layout object. In other words, even if you tell the layout of a `Group` to use a virtual layout, it will still create all its children, visible or not, because `Groups` don't support virtualization.

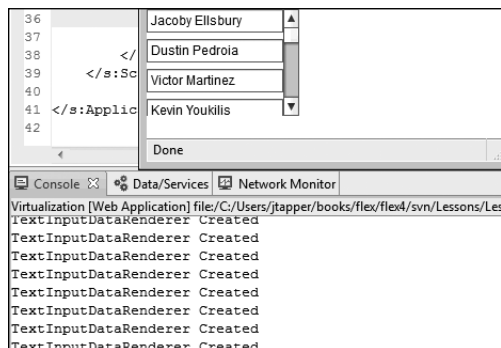
Implementing Virtualization

In this exercise, you'll take an existing application that has 25 items in a `dataProvider` of a `DataGroup`, but has room to show only four items at a time, and instruct it to use virtualization.

- 1 Import the `Virtualization.fxp` from the `Lesson10/independent` directory.

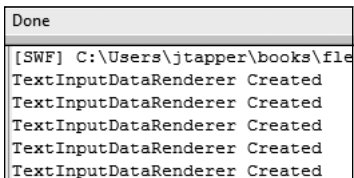
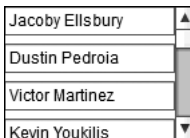
In the `VirtualizedVGroup.mxml` file in the default package of the `src` directory, you'll find an application that contains a `DataGroup` with 25 items in its `dataProvider` and that uses a variation on the `TextInputRenderer` you created earlier in this lesson.

- 2 Run the `Virtualization` application in Debug mode. Notice in the Console that there are 25 trace statements, one from the `creationComplete` event of each of the `itemRenderers`.



As you scroll through the items, you'll find you can never see more than five items at any one time (and most times only four items are visible at a time). However, as you can clearly see in the Console, there are far more than five instances of the `TextInputDataRenderer` created.

- 3 Find the instantiation of the `VerticalLayout`, and add the attribute `useVirtualLayout="true"`. Save and debug the application again. Notice this time there are only five trace statements of the `TextInputDataRenderer` instantiated.



Now you can see the real power of virtualization. Rather than having to create an instance of the renderer for each item in the `dataProvider`, which would be 25 total renderers, only 5 are created, as that is the most that can be seen in the control at any one time. There is no need to create and keep an additional 20 items in memory; instead, the same five renderers will be used to render whichever items need to be seen at any given time.

Virtualization with Lists

With the `List` class, virtualization is enabled automatically, so you do not need to explicitly tell the layout class to use `useVirtualLayout`. That much is assumed. In addition to virtualization, Lists also add selectability. *Selectability* is the idea that the user will be presented with several items and be allowed to choose one or more of them. Lists provide a series of properties, methods, and events surrounding the ideas of selectability. For instance, the `selectedIndex` and `selectedItem` properties allow you to specify or retrieve what is currently selected in the list.

In this exercise, you'll build a renderer to display the various categories shown in the top navigation of the application and specify the list displaying the categories to use that new renderer.

- 1 Open the FlexGrocer project.
- 2 Right-click the components folder, and create a new MXML component named **NavigationItem**. Specify the layout to be `VerticalLayout`, and the base class to be `spark.components.supportClasses.ItemRenderer`. Remove the height and width values.

`ItemRenderer` is a subclass of `DataRenderer`, which additionally implements the methods specified by the `ItemRenderer` interface. These include properties and methods related to displaying which items are and are not selected in a list.

- 3 Add an `Image` tag, specify a height of 31 and a width of 93. Set the source of the image to be `assets/nav_{data.name.toLowerCase()}.jpg`.

```
<s:Image
    source="assets/nav_{data.name.toLowerCase()}.jpg"
    height="31" width="93"/>
```

If you look in the assets directory, you'll find six files, with names such as `nav_dairy.jpg`, `nav_deli.jpg`, and so on. You may notice that the six names are very similar to the names of the categories from the `category.xml` file, with the difference that the names of the categories in the XML start with an uppercase letter, and in the filenames the categories start with a lowercase letter. To compensate for the difference of the upper- to lowercase letters, invoking the `String` class's `toLowerCase()` method forces the name to be all lowercase, so it can match the case of the file names. After the `toLowerCase()` method, the category that has a name of `Dairy` is lowercased and is concatenated into `nav_dairy.jpg`.

- 4 After the `Image`, add a `Label` whose text is bound to the `name` property of the data object.

```
<s:Label text="{data.name}"/>
```

In addition to the image, the desire is to show the category name below the image.

- 5 Find the `VerticalLayout` instantiation, and add a `horizontalAlign="center"` attribute.

```
<s:layout>
    <s:VerticalLayout horizontalAlign="center"/>
</s:layout>
```

Specifying a `horizontalAlign` of `center` will align the image and label horizontally to each other. You now have a functioning renderer that you can use in a `List` class to display the various categories.

- 6 Switch back to `FlexGrocer.mxml`.

The `List` displaying the categories is instantiated in the main application, `FlexGrocer.mxml`.

- Remove the `labelField` attribute from the instantiation of the `List` in the `controlBarContent`. Replace that attribute with the `itemRenderer` for this `List` to be your newly created `NavigationItem` class. Change the `height` property of the `List` to 52 to compensate for the larger size of the image and text.

```
<s:List left="200" height="52"
  dataProvider="{categoryService.categories}"
  itemRenderer="components.NavigationItem">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
</s:List>
```

- Save and run the application. It should now render the images and labels appropriately.



Displaying Grocery Products Based on Category Selection

You just passed a dataset to a `List` control and had an item display for each object in the dataset. At some point you'll also want to filter the collection of products to show only the products matching the selected category.

Displaying Grocery Items Based on Category

The first step will be to create a filter function in the `ProductService` class, which will accept a category `id` and filter the collection to show only the matching products.

- Open the `ProductService` class you created in Lesson 9, "Breaking the Application into Components."
- Create a private variable named `selectedCategory`, with a data type of `Number`, and a default value of 1.
- Create a private function named `filterForCategory()` that accepts a `Product` as an argument and returns a `Boolean`. In the body of the function, return a `Boolean` indicating whether the `catID` of the argument matches the `selectedCategory` property.

```
private var selectedCategory:Number=1;

private function filterForCategory( item:Product ):Boolean{
  return item.catID == selectedCategory;
}
```


- 4** In the `handleProductResult()` method, after the `products` `ArrayCollection` is instantiated, specify a `filterFunction()` of the `products` property to use your new `filterForCategory()` method. Next refresh the `products` collection.

```
products.filterFunction = filterForCategory;
products.refresh();
```

Now, when the collection is created, the `filterForCategory()` method is specified as its filter function, and the collection is refreshed, so the filter function will rerun.

- 5** Lastly, create a public function named `filterCollection()` that accepts a numeric argument, named `id`. Inside the function set the `id` as the value of the `selectedCategory` property, and then refresh the collection.

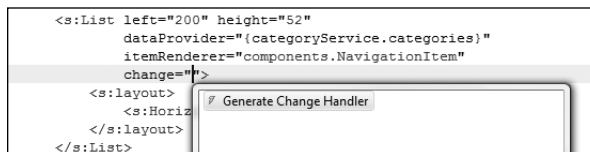
```
public function filterCollection( id:Number ):void{
    selectedCategory = id;
    products.refresh();
}
```

You now have everything you need in place to filter the collection to a specific category. All that remains is to call the `filterCollection()` method whenever the category changes.

Adding a Change Handler to the Category List

When the user selects an item from a list, a change event is broadcast, indicating that the selected item in the list is no longer the same. In this exercise, you'll handle the change event, and pass the `id` of the selected category to the `ProductService` to filter the collection so that only matching products are shown.

- 1 Open `FlexGrocer.mxml`.
- 2 Find the `List` class in the `controlBarContent`. Add a change handler to the `List`. Allow code completion to generate a change handler for you.



This will create a method named `list1_changeHandler()` for you, which accepts an argument named `event`, of type `IndexChangedEvent`. This method will automatically be set as the change handler for your list.

```
protected function list1_changeHandler(event:IndexChangedEvent):void
{
    // TODO Auto-generated method stub
}
```

- 3 Replace the `// TODO` auto-generated method stub of the `list1_changeHandler()` with a call to the `filterCollection()` method of the `productService`, passing in the `id` of the selected item from the list (`event.target.selectedItem.categoryID`).

```
protected function list1_changeHandler(event:IndexChangedEvent):void
{
    productService.filterCollection( event.target.selectedItem.categoryID );
}
```

- 4 Save and run the application.

Now, as you select products from the top category list, the products displayed in `ShoppingView` are updated accordingly.

What You Have Learned

In this lesson, you have:

- Populated a List control with a dataset (pages 242–245)
- Used a `DataGroup` with a dataset to display information with an `itemRenderer` (pages 245–246)
- Created an `itemRenderer` (pages 246–253)
- Learned about virtualization (pages 253–257)
- Responded to a user's choice from a list (pages 257–259)

LESSON 11

What You Will Learn

In this lesson, you will:

- Understand the benefits of loosely coupled architecture
- Dispatch events
- Declare events for a component
- Identify the need for your own event classes
- Create event subclasses
- Create and use a `UserAcknowledgeEvent` class
- Create and use a `ProductEvent` class
- Use event bubbling
- Use `ProductEvent` to add and remove a product
- Use the `CollectionEvent` to update the shopping cart total

Approximate Time

This lesson takes approximately 2 hours to complete.

LESSON 11

Creating and Dispatching Events

In previous lessons, you worked with events from built-in objects, such as the clicking of a Button or the changing of a List. You may remember that different events all descend from the same Event class but can have more specific information, such as the Mouse position in the MouseEvent. As you get deeper into application development, you will often need to dispatch events that contain your own information. In this lesson, you'll learn how to create an event object, set the metadata for the object, and dispatch it.

This lesson presents an overview of how to dispatch events within your application, and how to create new Event classes by creating a subclass of Event.



The shopping cart allows you to add and remove items.

Understanding the Benefits of Loose Coupling

At the end of Lesson 10, “Using DataGroups and Lists,” you were left without a way to add or remove items from the ShoppingCart. With your newly refactored application, the buttons for adding and removing are now inside the ProductItem class; however, the ShoppingCart for the whole application is defined within the ShoppingView class. This means that you can no longer directly call the `addItem()` and `removeItem()` methods of the ShoppingCart instance.

Technically, it would be possible to make the `shoppingCart` property public and still access the ShoppingCart instance from the ProductItem through an expression like this:

```
this.parent.parent.shoppingCart
```

However, such an expression can be very problematic for maintaining and debugging the application. During development, refactoring components is often desirable and sometimes essential. If you decide that the DataGroup should be inside another component, perhaps one responsible for all product display functions, the expression above may need to change as well.

Over the course of application development, one of two things tends to happen when using these types of expressions. Either you devote an increasing amount of time to maintaining the expressions as the application changes, which slows down progress and makes your day-to-day work increasingly frustrating. Or, worse yet, you stop refactoring your components even when it makes sense to do so. Maintaining the expressions becomes such a frustrating experience that you simply decide you’ll never change a specific area of code again. During active development this often leads to workarounds and suboptimal code, and can ultimately increase development time and the number of bugs.

Both of these ends have a common starting point. An expression like the one above caused ProductItem to have a dependency on ShoppingView. This means that anytime ShoppingView changes, you need to also remember to change ProductItem manually. Creating these types of interdependencies among objects in your application is called *tight coupling*, or making a *tightly coupled* application. In tightly coupled applications, objects often directly modify or access each other’s properties, creating maintenance difficulties. While there will always be some dependency between objects in an application, you need to strive to ensure that those dependencies are appropriate.

It can be both amusing and useful to think of objects using real-world analogs. In the real world, most objects are *loosely coupled*, which makes tightly coupled examples hyperbolic and fun.

Consider a satellite navigation system that provides directions while you drive a car. You and the navigation system exist in a loosely coupled way. When the navigation system is on, it provides

events indicating whether you should turn or proceed straight. You interpret those events and, ideally, make a decision that it is safe to turn or that you want to proceed in a different direction. Ultimately, you decide whether to engage your muscles and turn the steering wheel.

In a very tightly coupled version of this same architecture, the navigation system would take control of your body, forcing your muscles to move as needed to direct you to the new location. As every person is a bit different, the navigation system would have the ability to calibrate the amount of force required per muscle for each individual person. Perhaps it would even need to know your diet or exercise schedule to monitor changes in your musculature to ensure consistent results.

The point is simply that the tightly coupled architecture often involves objects having too much information and interacting directly with the internals of other objects. Further, once you make that first concession to make something tightly coupled, you may start down a path of making more and more concessions to make your application work. A loosely coupled architecture strives to use the ideas of notification and interfaces to allow objects to manage themselves more successfully.

In application development, maintaining only appropriate coupling can lead to better code reuse, easier refactoring, and the ability to debug an application in parts as opposed to en masse.

Dispatching Events

To broadcast an event from a component, you need to use the `dispatchEvent()` method. This method is defined in the `flash.events.EventDispatcher` class. Some objects in Flex (for example, the `UIComponent`), descend directly from `EventDispatcher` and can dispatch events without any further work.

The following is the inheritance hierarchy of the `UIComponent` class:

```
mx.core.UIComponent extends
mx.core.FlexSprite extends
flash.display.Sprite extends
flash.display.DisplayObjectContainer extends
flash.display.InteractiveObject extends
flash.display.DisplayObject extends
flash.events.EventDispatcher
```

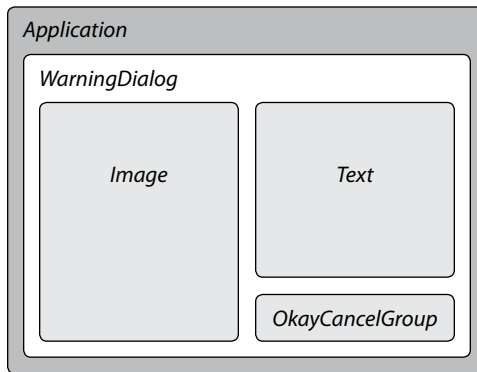
The `dispatchEvent()` method takes a single argument, which is an event object to be dispatched. When an event is dispatched, anything listening for that event is notified, and any event listeners (handlers) are executed. This offers a way to create a loosely coupled architecture.

- 1 Import the EventLab.fxp from the Lesson11/independent folder into Flash Builder. Please refer to the appendix for complete instructions on importing a project.
- 2 Open and run EventLab.mxml.



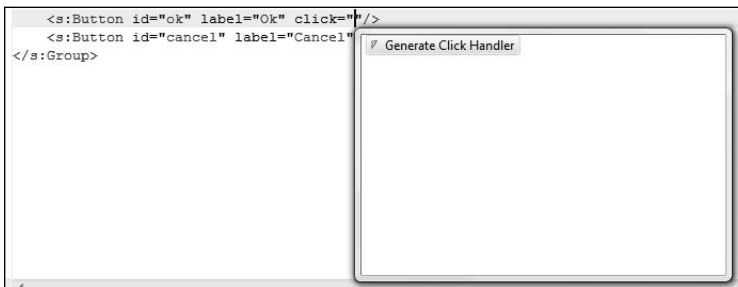
The application displays a simple warning dialog box with an image, text, and Ok and Cancel buttons. You will use this dialog box throughout this lesson, adding events and eventually making the Ok and Cancel buttons function properly.

The EventLab application contains two custom components: `WarningDialog.mxml` and `OkayCancelGroup.mxml`. These custom components are nested inside the Application, as the following diagram demonstrates:



The `WarningDialog` is directly inside the `Application`. It contains an image, text about the warning, and a custom component named `OkayCancelGroup`. The Ok and Cancel buttons are so often reused together in an application that they have been put into a custom component for this lab.

- 3 Close your browser and open the `OkayCancelGroup.mxml` class from the components package.
- 4 Find the MXML tag for the Ok button. Inside that tag, begin typing the word **click**. After the third letter, Flash Builder will understand that you intend to handle the `click` event. Press Enter, and Flash Builder will complete the word `click` and add the equal sign and quotes. Flash Builder will now prompt you to Generate Click Handler. Either click the option with your mouse or press Enter.



Flash Builder will generate the `<fx:Script>` block along with a new function named `ok_clickHandler()`, which accepts a `MouseEvent`.

- 5 Repeat step 4 for the Cancel button. When complete, the relevant portion of your `OkayCancelGroup` should read as follows:

```
<fx:Script>
  <![CDATA[
    protected function ok_clickHandler(event:MouseEvent):void
    {
      // TODO Auto-generated method stub
    }

    protected function cancel_clickHandler(event:MouseEvent):void
    {
      // TODO Auto-generated method stub
    }

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

```
<s:Button id="ok" label="Ok" click="ok_clickHandler(event)"/>
<s:Button id="cancel" label="Cancel" click="cancel_clickHandler(event)"/>
```


Flex is a general-purpose component framework, so it dispatches general-purpose events. For example, buttons in Flex dispatch a `click` event when they are clicked. That is a wonderful starting point, but the concept of a click doesn't mean much in the context of your application.

Ultimately, when a button is clicked, it is likely to mean something specific. In this component, when the Cancel button is clicked, it means the user wants to cancel the operation. When the Ok button is clicked, it means the user acknowledges the issue and wants to proceed.

Therefore, for the purposes of writing code that is much more legible, maintainable, and easier to debug, you often handle events like the `click` event inside a component, and then immediately dispatch a new, more specific event that makes sense within the context of the application.

Inside the `ok_clickHandler()` method, delete the `//TODO` comment and create a new instance of the `Event` object, with the type `userAccept`.

```
var e:Event = new Event("userAccept");
```

This creates the new instance of the `Event` object, which will be dispatched to notify the remainder of the application of the user's choice.

- 6 Just after creating the event object, dispatch it.

```
this.dispatchEvent(e);
```

This dispatches the event so that any listening components can respond as needed.

- 7 Repeat these steps in the `cancel_clickHandler()` method, dispatching an event with the type `userCancel`.

```
var e:Event = new Event("userCancel");  
this.dispatchEvent( e );
```

- 8 Save your component and open `WarningDialog.mxml` from the `components` package.
- 9 Find the instance of the `OkayCancelGroup` component. Inside this tag, inform Flex that you wish to handle the `userAccept` event with a method named `handleAccept()`, passing the event object to that method.

```
<components:OkayCancelGroup id="okCancelGroup"  
    left="219" bottom="22" userAccept="handleAccept(event)"/>
```

Note that you will not get the convenient code hinting, and Flash Builder will not be able to generate an event handler for you at this time. In fact, if you save this file presently, you will see an error in the Problems view. That is all right for now. You will examine these issues soon and fix them in the next exercise.

- 10** Add a new method named `handleAccept()` to the Script block of `WarningDialog.mxml`. The method will accept a single parameter named `event` of type `Event`.

```
private function handleAccept( event:Event ):void {  
  
}
```

- TIP:** You may notice that when you're asked to create a function in this book, it's usually private. However, when Flex creates functions for you, they are usually protected. Both private and protected functions can be accessed by code within a given class, but not by other types of objects in the system. However, when dealing with inheritance, protected functions can be used by subclasses of your objects, whereas private functions cannot. As Flash Builder doesn't know how your functions will be used, it simply defaults to a more permissive setting.

- 11** Save `WarningDialog.mxml`. Look at the Problems view and notice the error.

The Problems view is now showing an error: `Cannot resolve attribute 'userAccept' for component type components.OkayCancelGroup.`

While you are dispatching an event in your `OkayCancelGroup` named `userAccept`, the Flex compiler, as of yet, is unaware of it.

For the compiler to know what `userAccept` means, you need to perform one additional step, adding metadata to the component that specifically declares any events the component will dispatch. This will also enable code-completion and handler generation in Flash Builder.

Declaring Events for a Component

Every component must explicitly declare the events it can dispatch. Components that are subclasses of other components can also dispatch any events that their superclasses have declared. In Flex, events are declared with metadata tags. This is done with the `[Event]` metadata, which is used to declare events publicly so that the MXML compiler can verify that the user did not simply make a typo. In MXML, an event declaration looks like this:

```
<fx:Metadata>  
    [Event(name="userAccept" ,type="flash.events.Event")]  
</fx:Metadata>
```

The `<fx:Metadata>` tag declares that the child elements are all metadata. Next, any metadata is declared. Notice that the tags are enclosed within square brackets. Details for these tags are defined within parentheses. In this example, you can see a `userAccept` event declared. This event will be an instance of the `flash.events.Event` class. In this exercise, you will fix the error from the previous exercise by declaring a custom event for the `OkayCancelGroup` component.

- 1 Open OkayCancelGroup.mxml from your components package.
- 2 Before the `<fx:Script>` block, add a metadata block to declare the `userAccept` event.

```
<fx:Metadata>
    [Event(name="userAccept", type="flash.events.Event")]
</fx:Metadata>
```

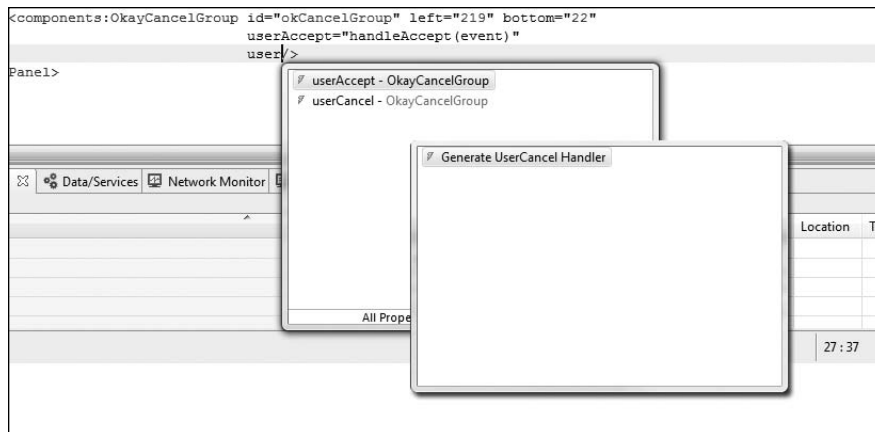
If the type is omitted, Flash Builder will assume it is an instance of the `flash.events.Event` class. While in this case it might save you a few keystrokes, it is usually best to declare the type each time you create a new Event declaration to ensure completeness and provide additional documentation.

- 3 Directly below the first Event declaration, but inside the same metadata tag, add a second Event declaration for the `userCancel` event.

```
<fx:Metadata>
    [Event(name="userAccept", type="flash.events.Event")]
    [Event(name="userCancel", type="flash.events.Event")]
</fx:Metadata>
```

Save `OkayCancelGroup.mxml`. The errors should now be gone, as Flash Builder understands that this component will be dispatching the named events.

- 4 Return to the `WarningDialog.mxml` file and find the `OkayCancelGroup` tag again.
- 5 You will now handle the `userCancel` event. Begin typing `userCancel`. You will see that Flash Builder now also understands that this component dispatches the `userAccept` and `userCancel` events and offers code hinting. Choose the `userCancel` event and then choose `Generate UserCancel Handler`.



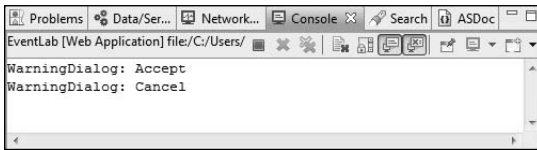
Flash Builder will add a method named `okCancelGroup_userCancelHandler()` to your component. The method will accept a parameter of type `Event` automatically because, due to your metadata declaration, Flash Builder knows what type of event to expect.

- 6 Add a trace statement to the `handleAccept()` method that traces the words *WarningDialog: Accept* to the console.
- 7 Add a trace statement to the `okCancelGroup_userCancelHandler()` method that traces the words *WarningDialog: Cancel* to the console. Your code should read as follows:

```
private function handleAccept( event:Event ):void {
    trace( "WarningDialog: Accept" );
}

protected function okCancelGroup_userCancelHandler(event:Event):void {
    trace( "WarningDialog: Cancel" );
}
```

- 8 Debug your application. When you click the Ok or Cancel buttons, you should see the corresponding text traced out to the Console view.



- 9 Terminate your debugging session and return to Flash Builder.

You now have a simple reusable component, capable of communicating with other areas of the application in a loosely coupled manner.

Identifying the Need for Custom Event Classes

In the previous exercise, events notified other parts of the application about a user action. In addition to notifications, you sometimes need to pass data with events. The `flash.events.Event` class supports only the properties needed for the most basic style of event, but you are always free to subclass events to make more specific types.

Passing data with events is a common practice in Flex and can be extremely beneficial. In this example, you will record a timestamp each time a user clicks Ok or Cancel. This timestamp may be used to log data later, but it's very important that you record the exact time when the user clicks the button. To do that, you are going to create your own event class that will contain this time information.

Earlier in this lesson and others, you used the `dispatchEvent()` method to broadcast an event from a component. The `dispatchEvent()` method accepts an instance of the `flash.events.Event` class as its only parameter. Therefore, any events you wish to dispatch must be subclasses of this `Event`.

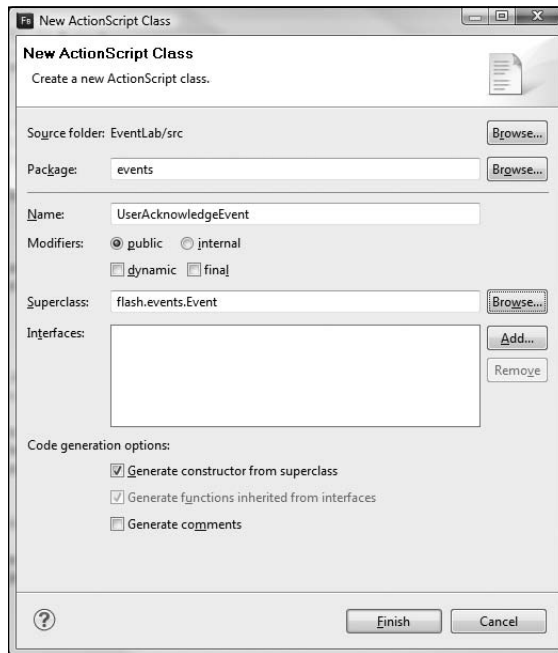
You can add any methods or properties to your event, but you are required to override one method each time you create a new `Event`. This method, named `clone()`, is responsible for creating a new event based on the properties of your original event. In other words, it creates an identical clone of it.

This method is used by the Flex framework in a number of places, including event bubbling, a concept you will understand before the end of this lesson.

Building and Using the `UserAcknowledgeEvent`

You are about to create an event subclass. This event will have an additional `timestamp` property, allowing your application to note the exact time when the button was clicked.

- 1 Right-click the `src` folder of the `EventLab` project and create a new ActionScript class. Specify **events** as the Package name and **UserAcknowledgeEvent** as the Name of the class. Set the Superclass to **flash.events.Event**.



Filling out the dialog box automatically creates the skeleton of the class seen here:

```
package events {
    import flash.events.Event;
    public class UserAcknowledgeEvent extends Event {
        public function UserAcknowledgeEvent(type:String,
            ► bubbles:Boolean=false, cancelable:Boolean=false) {
            super(type, bubbles, cancelable);
        }
    }
}
```

- 2 Inside the class definition, create a public property named `timestamp` to hold a `Date` instance.

```
public var timestamp:Date;
```

ActionScript doesn't have a time data type. However, the `Date` object can store both date and time information.

- 3 Change the constructor to accept only two parameters: `type`, which is a `String`, and `timestamp`, which is a `Date`. In the constructor, pass `type` to the superclass and store the `timestamp` in the instance variable you created in step 2.

```
public function UserAcknowledgeEvent(type:String, timestamp:Date) {
    super(type);
    this.timestamp = timestamp;
}
```

Like all constructors in ActionScript 3.0, this one is public. The two arguments will be used to populate the event. The `timestamp` property will be used to hold the time when the event occurred. The `type` property defines the type of action that occurred to trigger this event (for example `userAccept` or `userCancel`). Events often accept two other optional parameters, which you just deleted. We will explore one of those optional parameters later in this lesson.

- 4 Override the `clone()` method. In this method, you will return a new instance of the `UserAcknowledgeEvent` with the same values.

```
override public function clone():Event {
    return new UserAcknowledgeEvent( type, timestamp );
}
```

When you override a method in ActionScript 3.0, the method must be defined with exactly the same parameters and return type of the superclass and must include the `override` keyword. Therefore, the `clone()` method needs to be defined as public, it must take no parameters, and it must return an instance of the `Event` class. Your new event is a subclass of the `Event` class and can therefore be returned by this method.

The complete `UserAcknowledge` class should look like the following code block:

```
package events {
    import flash.events.Event;

    public class UserAcknowledgeEvent extends Event {
        public var timestamp:Date;
        public function UserAcknowledgeEvent(type:String, timestamp:Date) {
            super(type);
            this.timestamp = timestamp;
        }

        override public function clone():Event {
            return new UserAcknowledgeEvent( type, timestamp );
        }
    }
}
```

- 5 Open the `OkayCancelGroup.mxml` file from your components package.
- 6 Inside the `<fx:Script>` block, find the `ok_clickHandler()`. Currently the method dispatches an instance of the `Event` class. Change this method to instantiate a new instance of the `UserAcknowledgeEvent` class. Pass `userAccept` as the first parameter and a new instance of the `Date` object as the second. Then dispatch this new event.

```
protected function ok_clickHandler(event:MouseEvent):void {
    var e:UserAcknowledgeEvent =
        ➤ new UserAcknowledgeEvent("userAccept", new Date() );
    this.dispatchEvent( e );
}
```

Each time you create a new `Date` object, it defaults to the current `Date` and `Time`. If you used the code-completion feature, an import for `UserAcknowledgeEvent` was added to your class automatically; otherwise, you will need to manually add the import.

```
import events.UserAcknowledgeEvent;
```

- 7 Repeat this process for `cancel_clickHandler()`. Instantiate a new instance of the `UserAcknowledgeEvent` class, passing `userCancel` as the first parameter and a new instance of the `Date` object as the second. Then dispatch this new event.

```
protected function cancel_clickHandler(event:MouseEvent):void {
    var e:UserAcknowledgeEvent =
        ➤ new UserAcknowledgeEvent("userCancel", new Date());
    this.dispatchEvent( e );
}
```

You are now dispatching a `UserAcknowledgeEvent` each time a button is clicked, however, the metadata for this class still indicates that you are dispatching generic Flash events. Change the metadata to indicate that each of these events will now dispatch an `events.UserAcknowledgeEvent` instance.

```
<fx:Metadata>
    [Event(name="userAccept" ,type="events.UserAcknowledgeEvent")]
    [Event(name="userCancel" ,type="events.UserAcknowledgeEvent")]
</fx:Metadata>
```

Your `OkayCancelGroup` is now broadcasting `UserAcknowledgeEvent` instances in all cases. All your code will work properly at this point, and you can run it now to verify.

While event times are being stored, you will need to update your `WarningDialog.mxml` file to output this new information.

- 8 Open the `WarningDialog.mxml` file from your components package.
- 9 Find the `handleAccept()` method and change the type of the event parameter to `events.UserAcknowledgeEvent`.

```
private function handleAccept( event:UserAcknowledgeEvent ):void {
    trace( "WarningDialog: Accept" )
}
```

If you used the code-completion feature, an import for `UserAcknowledgeEvent` was added to your class automatically; otherwise, you will need to manually add the import.

Previously this method accepted an instance of the `Event` class. Remember that your `UserAcknowledgeEvent` is a type of `Event`, so this code will still work properly. However, Flex knows that the `timestamp` property does not exist in the `Event` class. So, to use your new `timestamp`, this method must be updated to the appropriate type.

- 10 Update the `trace` statement by adding a space after the word *Accept* and before the quotation mark. Then concatenate the `event.timestamp` to the end of the output.

```
private function handleAccept( event:UserAcknowledgeEvent ):void {
    trace( "WarningDialog: Accept " + event.timestamp );
}
```

- 11 Repeat this process for `cancel`, changing the event to type `UserAcknowledgeEvent`, adding a space after the word *Cancel* and then concatenating the `event.timestamp` to the end of the output.

```
protected function okCancelGroup_userCancelHandler(event:UserAcknowledgeEvent):
    void {
    trace( "WarningDialog: Cancel " + event.timestamp );
}
```


- 12** Save and debug the application. It should now output the trace statements along with the time of each click.

```
WarningDialog: Accept Wed Jul 28 15:59:26 GMT-0600 2010
```

```
WarningDialog: Cancel Wed Jul 28 15:59:26 GMT-0600 2010
```

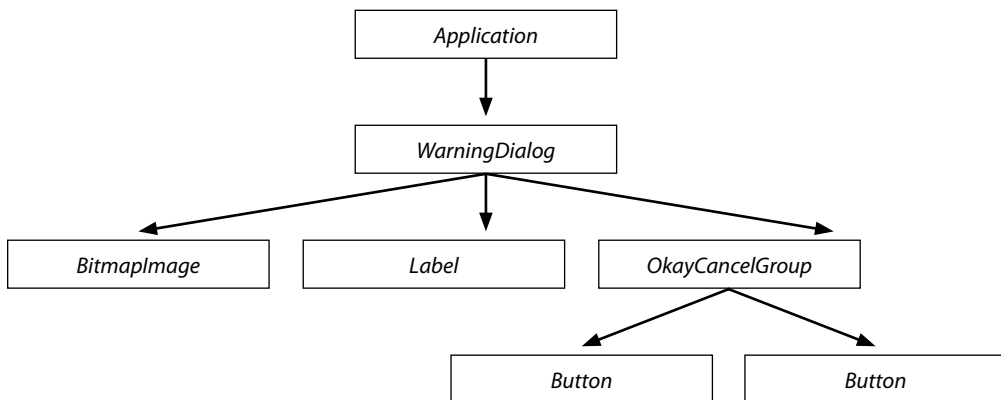
- 13** Terminate the debugging session.

Understanding Event Flow and Event Bubbling

There are two broad categories of classes you have worked with so far in Flex: classes that inherit from `DisplayObject` and classes that do not. All the Flex UI components such as Lists, Buttons, and Labels are `DisplayObjects`. The value objects you created, as well as classes like `HTTPService`, which do not have a visual display, are not `DisplayObjects`.

A lot of work is done in Flex to classes that appear on the screen. The Flash Player and Flex framework need to size and position them on the screen, ensure that they are created at the correct time, and eventually draw them on your monitor. Therefore Flash Player maintains a list, called the *display list*, of every visual component currently available to the user. So far, every time you have created a visual component, it has been added to the display list. In Lesson 14, “Implementing Navigation,” you will learn to have a little more control over when this occurs.

As a mental model, the display list can be thought of as a hierarchy, or tree. Each time something is added to the list, it is added relative to its parent. So, a partial display list for the EventLab application looks like this:



This is only a partial list, as in reality things like Buttons are actually further composed of a label and a background, and so on. Only visual elements are added to the display list, so objects without a display, like the `HTTPService`, will not appear on this list. This is important, because once something is on the display list, Flash Player provides additional functionality when dispatching events.

If the event target (the object dispatching the event) is not a visual element, Flash Player simply dispatches the event object directly to the designated target. For example, Flash Player dispatches the `result` event directly to an `HTTPService` component.

However, if the target is a visual element on the display list, Flash Player dispatches the event, and it travels from the outermost container (the Application container in our simplified example), down through the target component, and optionally back up to the Application container.

Event flow is a description of how that event object travels through an application. As you have seen by now, Flex applications are structured in a parent-child hierarchy, with the Application container being the top-level parent. Earlier in this lesson, you also saw that `flash.events.EventDispatcher` is the superclass for all components in Flex. This means that all visual objects in Flex can use events and participate in the event flow; they can all listen for an event with the `addEventListener()` method and dispatch their own events.

During this trip from the Application to the component that was responsible for the event (known as the *target* of the event) and optionally back to the Application, other components within the event's path may listen for and act on the event. In fact, components can decide if they want to listen on the way to the object or on the way back to the Application.

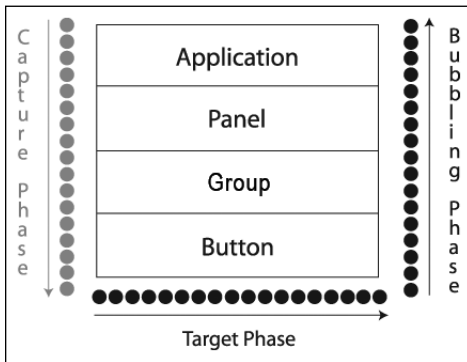
The event flow is conceptually divided into three parts:

- The *capture phase* comprises all the components on the trip from the base application to the parent of the event's target. In other words, everything from the application to the target, not including the target itself.
- The *target phase* occurs when the event reaches the target.
- The *bubbling phase* comprises all the components encountered on the return trip from the target back to the root application.

The following image describes a branch of an application in which a Button is contained within a Group, which is contained by a Panel, which sits in the root Application. For the context of this example, other elements in the application are moot.

Application
Panel
Group
Button

If a user clicks the Button, Flash Player dispatches an event object into the event flow. The object's journey starts at the Application, moves down to the Panel, moves to the Group, and finally gets to the Button. The event object then "bubbles" back up to Application, moving again through the Group and Panel on its way up.



In this example, the capture phase includes the Application, Panel, and Group during the initial downward journey. The target phase comprises the time spent at the Button. The bubbling phase comprises the Group, Panel, and Application containers as they are encountered during the return trip.

All instances of the Event class have a `bubbles` property that indicates whether that event object will participate in the bubbling phase of the event flow. If this property is set to `true`, the event makes a round-trip; otherwise it ends when the target phase is complete.

All this means that an event can occur in a child component and be heard in a parent. Consider this simple example:

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

```

        click="showAlert(event)">
<fx:Script>
  <![CDATA[
    import mx.controls.Alert;
    private function showAlert( event:Event ):void {
      var msg:String = event.target.toString() +" clicked";
      Alert.show( msg );
    }
  ]]>
</fx:Script>
<s:Panel id="panel"
  click="showAlert(event)" >
  <s:Group id="group"
    click="showAlert(event)" >
    <s:Button id="button"
      click="showAlert(event)"/>
    </s:Group>
  </s:Panel>
</s:Application>

```

In this case, there is a Button control inside a Group, inside a Panel, inside an Application. When the button is clicked, the `click` event of the Button control is heard from the event handler of the Button, Group, Panel, and Application, and therefore four Alert boxes pop up, all saying the following:

```

TestApp.ApplicationSkin2._ApplicationSkin_Group1.contentGroup.panel.PanelSkin6.
  ➤ _PanelSkin_Group1.contents.contentGroup.group.button clicked

```

This string represents the entire path the event traveled in its journey to dispatch. The `click` event of the Button control can be captured at the Button control itself or in any of the parent containers of the Button instance. This happens because `click` is a bubbling event. The `bubbles` property of the Event class is Boolean, which indicates whether an event should bubble. By default, `bubbles` is set to `false` on newly created events (although it is preset to `true` for some built-in events, such as `click`; you can check the API docs to find this information as it pertains to any event). When you create event instances or event subclass instances, you can decide whether you want to enable bubbling for the event. If you leave the bubbling to the default `false` value, the event can be captured only at the source of the event (the Button control in the preceding example). However, if bubbling is set to `true`, the event can be captured by a parent of the dispatching component (such as the Group, Panel, and Application).

Currently, the `OkayCancelGroup` class dispatches an event, and that event is being used by the `WarningDialog`. Next you will make that event bubble so that it can be handled in the Application itself.

1 Open `UserAcknowledgeEvent.as` from the events package.

2 Locate the constructor and the call to the superclass.

```
public function UserAcknowledgeEvent(type:String, timestamp:Date) {
    super(type);
    this.timestamp = timestamp;
}
```

The `flash.events.Event` constructor can accept up to three arguments. The first is the type, which you are passing presently. The second is a Boolean flag indicating whether the event should bubble, and the third is a Boolean indicating whether the event is cancelable (whether another object can cancel the event).

3 Pass `true` to the `bubbles` argument of the superclass.

```
public function UserAcknowledgeEvent(type:String, timestamp:Date) {
    super(type, true);
    this.timestamp = timestamp;
}
```

This event will now make the return trip (bubble) from the `OkayCancelGroup` all the way back up to the `Application`.

4 Open `WarningDialog.mxml`.

The event is now going to pass through this class on the way back to the `Application`. Therefore, if you choose, you can act on the event here as well.

5 Add an `<fx:Metadata>` tag and the appropriate event metadata to the `WarningDialog` indicating that this object will also dispatch a `userAccept` and `userCancel` event, both of type `events.UserAcknowledgeEvent`.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Panel xmlns:fx="http://ns.adobe.com/mxml/2009"
xmlns:s="library://ns.adobe.com/flex/spark"
width="400" height="225"
xmlns:components="components.*"
title="Warning: Something to be warned about!">

    <fx:Metadata>
        [Event(name="userAccept", type="events.UserAcknowledgeEvent")]
        [Event(name="userCancel", type="events.UserAcknowledgeEvent")]
    </fx:Metadata>

    <fx:Script>
        ...
    </fx:Script>
    ...
</s:Panel>
```

Feel free to copy the entire metadata block from the `OkayCancelGroup` if you wish to save a few keystrokes.

6 Open `EventLab.mxml`.

You can now listen for either of these events on the `WarningDialog` as well as the `OkayCancelGroup`, as they will bubble up the display list.

7 Inside the `WarningDialog` tag, begin typing `userAccept` and `userCancel`. If the metadata in step 5 was added correctly, Flash Builder will offer you code completion and the ability to generate an event handler. Accept this offer, creating an event handler for both events.

```
<components:WarningDialog id="warningDialog"
    horizontalCenter="0" verticalCenter="0"
    userAccept="warningDialog_userAcceptHandler(event)"
    userCancel="warningDialog_userCancelHandler(event)"/>
```

8 Inside the `warningDialog_userAcceptHandler()`, add a `trace` statement to output the class name, the event, and the timestamp as follows:

```
protected function warningDialog_userAcceptHandler(event:UserAcknowledgeEvent):
    void {
        trace( "EventLab: Accept " + event.timestamp );
    }
```

9 Inside the `warningDialog_userCancelHandler()`, also add a `trace` statement to output the class name, the event, and the timestamp as follows:

```
protected function warningDialog_userCancelHandler(event:UserAcknowledgeEvent):
    void {
        trace( "EventLab: Cancel " + event.timestamp );
    }
```

10 Debug the `EventLab` application and click both the `Ok` and `Cancel` buttons.

```
WarningDialog: Accept Sun Feb 7 13:42:37 GMT-0600 2010
EventLab: Accept Sun Feb 7 13:42:37 GMT-0600 2010
WarningDialog: Cancel Sun Feb 7 13:42:37 GMT-0600 2010
EventLab: Cancel Sun Feb 7 13:42:37 GMT-0600 2010
```

Notice that the `trace` statement first occurs in the `WarningDialog` where the event is received, followed by the `EventLab`. You have created a bubbling event and handled it in the application. You are now ready to fix the product addition and removal in the `FlexGrocer` application.

11 Terminate the debugging session and close the `EventLab` project.

You will now move back to the `FlexGrocer` application and apply this procedure there.

Creating and Using the ProductEvent Class

In this next exercise, you'll create an event subclass called `ProductEvent`. `ProductEvent` will add a single property to the `Event` class named `product`, which will hold an instance of your `Product` value object. You will then refactor the `ProductItem` based on your new knowledge of events to reduce some application coupling and restore functionality.

- 1 Open the `FlexGrocer` project.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer.fxp` project from the `Lesson11/start` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Right-click the `src` folder and create a new `ActionScript` class. Set the Package of the new class to **events**. Name the new class **ProductEvent**, and set **flash.events.Event** as the superclass.

The skeleton for your new class should look like this:

```
package events {
    import flash.events.Event;
    public class ProductEvent extends Event {
        public function ProductEvent(type:String, bubbles:Boolean=false,
            ↪ cancelable:Boolean=false) {
            super(type, bubbles, cancelable);
        }
    }
}
```

- 3 Create a new public property for your class, named `product`, with a data type `Product`.

If you use code hinting and choose the `Product` class from the list, the `import` statement for `valueObjects.Product` will be added automatically; if not, you will need to manually import the class.

- 4 Modify the constructor for your class so that it takes two arguments. The first argument will remain the `type`, which is a `String`. The second argument is an instance of the `Product` class.

```
public function ProductEvent(type:String, product:Product ) {
}
```

- 5** Inside the constructor, pass the type to the superclass, along with a true for the value of the bubbles parameter. Set your local product instance variable equal to the product argument of the constructor.

```
public function ProductEvent(type:String, product:Product ) {
    super(type, true);
    this.product = product;
}
```

- 6** Override the clone() method. This method will return a new instance of the ProductEvent class with the same type and product.

```
override public function clone():Event{
    return new ProductEvent(type, product);
}
```

- 7** Save the ProductEvent class. The class should currently look like this:

```
package events {
    import flash.events.Event;
    import valueObjects.Product;
    public class ProductEvent extends Event {
        public var product:Product;

        public function ProductEvent(type:String, product:Product ) {
            super(type, true);
            this.product = product;
        }

        override public function clone():Event {
            return new ProductEvent( type, product );
        }
    }
}
```

- 8** Open ProductItem.mxml from your components package.

- 9** Remove the public variable named shoppingCart.

You are no longer going to attempt to add and remove items from the shopping cart directly. Instead you will use events to inform other components of the user's actions.

- 10** Find the addToCart() method. Delete all the existing contents of this method.

- 11** Inside the addToCart() method, declare a new local variable named event of type ProductEvent, and set it equal to a new instance of the ProductEvent event class. For the type parameter of the ProductEvent constructor, pass the string *addProduct*. Then pass the product argument of this method as the second constructor parameter. Finally, dispatch the event.


```
private function addToCart( product:Product ):void {
    var event:ProductEvent = new ProductEvent( "addProduct", product );
    dispatchEvent( event );
}
```

If you use code-completion, `events.ProductEvent` will be imported for you. If not, be sure to import it manually.

- 12** Repeat this process for the `removeFromCart()` method, passing the string `removeProduct` to the `ProductEvent` type parameter.

```
private function removeFromCart( product:Product ):void {
    var event:ProductEvent = new ProductEvent( "removeProduct", product );
    dispatchEvent( event );
}
```

- 13** Add an `<fx:Metadata>` tag to this class. Inside it, declare that `ProductItem.mxml` will dispatch two events named `addProduct` and `removeProduct`. Indicate that both events will be of type `events.ProductEvent`.

```
<fx:Metadata>
    [Event(name="addProduct", type="events.ProductEvent")]
    [Event(name="removeProduct", type="events.ProductEvent")]
</fx:Metadata>
```

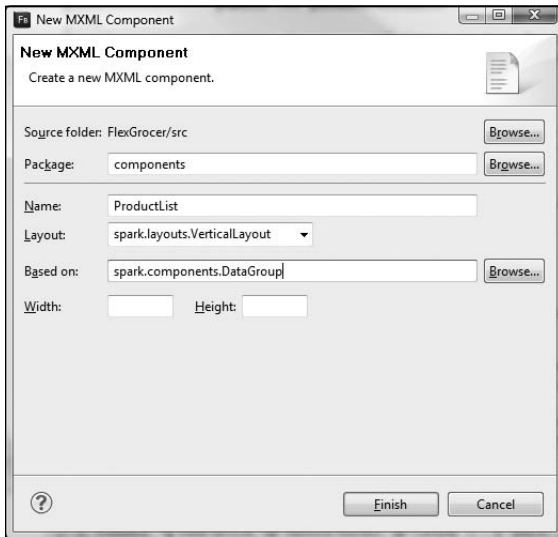
- 14** Save this class and ensure there are no problems in the Problems view.

You are now dispatching a bubbling event from the `ProductItem` when the Add To Cart or Remove From Cart buttons are clicked.

Creating a ProductList Component

As you learned in previous lessons, you create a custom component in Flex whenever you need to compose new functionality. Previously, you created a `DataGroup` that displayed products on the screen. While you still want to use that `DataGroup`, you now need a `DataGroup` that will dispatch `addProduct` and `removeProduct` events. Anytime you make a component's job more specific, you are talking about subclassing. In this exercise you will subclass `DataGroup` to make a `ProductList`. `ProductList` is a `DataGroup` with the extra event metadata needed by Flex.

- Right-click the components package and choose `New > MXML Component`. Ensure the package is set to the word **components** and set the Name to **ProductList**. Set the layout to **spark.layouts.VerticalLayout** and the "Based on" field to **spark.components.DataGroup**, and clear the Width and Height fields. Click Finish.



- 2 Set the `itemRenderer` property on the `DataGroup` node to `components.ProductItem`.

```
<?xml version="1.0" encoding="utf-8"?>
<s:DataGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  itemRenderer="components.ProductItem">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here-->
  </fx:Declarations>
</s:DataGroup>
```

The `DataGroup` in your `ShoppingView` uses `components.ProductItem` as an `itemRenderer`. This new `ProductList` is intended to replace that `DataGroup` with equivalent functionality plus events.

- 3 Add an `<fx:Metadata>` tag to this class. Inside it, declare that `ProductItem.mxml` will dispatch two events, named `addProduct` and `removeProduct`. Indicate that both events will be of type `events.ProductEvent`.

```
<fx:Metadata>
  [Event(name="addProduct", type="events.ProductEvent")]
  [Event(name="removeProduct", type="events.ProductEvent")]
</fx:Metadata>
```

This DataGroup is going to use the components.ProductItem renderer. As you declared earlier, that itemRenderer will dispatch two bubbling events: addProduct and removeProduct. As you saw in the EventLab, when an event bubbles, you can listen for the event on any of the parent instances. In this case, you will listen for the addProduct and removeProduct events on the ProductList.

- 4** Save the ProductList class. It should read as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<s:DataGroup xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  itemRenderer="components.ProductItem">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here-->
  </fx:Declarations>
  <fx:Metadata>
    [Event(name="addProduct", type="events.ProductEvent")]
    [Event(name="removeProduct", type="events.ProductEvent")]
  </fx:Metadata>
</s:DataGroup>
```

Using the ProductList Component

You will now replace the DataGroup in your ShoppingView with your new ProductList component.

- 1** Open the ShoppingView.mxml file and locate the DataGroup on approximately line 40.
- 2** Directly below the DataGroup, add a ProductList component.

If you used code-completion, Flash Builder automatically added a component name space on your behalf. If you did not, you will need to add this namespace manually.

```
<components:ProductList/>
```

- 3** Many of the properties on the DataGroup will be the same on your new ProductList. Copy the width, height, and visible properties (for both the normal and cartView state) to your ProductList tag.

```
<components:ProductList width="100%" height="100%"
  width.cartView="0" height.cartView="0"
  visible.cartView="false"/>
```

- 4** Finally, move the dataProvider property to the new ProductList and delete the DataGroup. Your new ProductList tag should look like the following code:


```
<components:ProductList width="100%" height="100%"
    width.cartView="0" height.cartView="0"
    visible.cartView="false"
    dataProvider="{groceryInventory}"/>
```

- 5 Save this file and run the application. You shouldn't receive any errors, and the Products should display as before.

Using ProductEvent to Add and Remove a Product

An instance of the ProductEvent class is bubbling up the display list each time the AddToCart button is clicked. You are now going to listen to that event and use it to actually add the product to the cart.

- 1 Open ShoppingView.mxml from the views package.
- 2 Inside the Script block, add a new private method named `addProductHandler()`. This function will accept a single parameter named `event` of type `ProductEvent` and return `void`.

 **TIP:** In this case you are writing the event handlers manually. When Flash Builder automatically creates an event handler on your behalf, it normally names it to correspond to the control that is using the event (so, something like `productList1_addProductHandler()` if the `ProductList` were using it). That is fine in most cases, but this particular handler is going to be used by multiple controls, so you are naming it manually.

- 3 Still inside the Script block, add another new private method named `removeProductHandler()`. This function will also accept a single parameter named `event` of type `ProductEvent` and return `void`.

```
private function addProductHandler(event:ProductEvent):void {
}
```

```
private function removeProductHandler(event:ProductEvent):void {
}
```

If you did not use code-completion, add the import for `events.ProductEvent` at this time. Again, you are making these methods private, as they are not needed outside this class.

- 4 Inside the `addProductHandler()` method, create a new local variable named `sci` of type `ShoppingCartItem`. Set this variable equal to a new instance of the `ShoppingCartItem` class, passing the `product` property of your event object to its constructor.


```
var sci:ShoppingCartItem = new ShoppingCartItem( event.product );
```

You already did the hard work by ensuring the event would have a reference to the clicked product available. Now you simply need to use it.

- 5 Still inside the `addProductHandler()` method, add the `ShoppingCartItem` instance to the shopping cart using the `addItem()` method of the `shoppingCart` reference. Your code should look like this:

```
private function addProductHandler(event:ProductEvent):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( event.product );
    shoppingCart.addItem( sci );
}
```

- 6 Duplicate this concept inside the `removeProductHandler()` method. Create a new local variable named `sci` of type `ShoppingCartItem` and assign it a new `ShoppingCartItem` instance with `event.product` passed to its constructor. However, in this case, call the `removeItem()` method of the `shoppingCart`, passing the local `sci` variable.

```
private function removeProductHandler(event:ProductEvent):void {
    var sci:ShoppingCartItem = new ShoppingCartItem( event.product );
    shoppingCart.removeItem( sci );
}
```

You now have two event handlers ready to add or remove products from the cart. You will now simply indicate that these two handlers should be used by your `ProductList` for this purpose.

- 7 Find the `ProductList` tag and indicate that you will handle the `ProductList`'s `addProduct` event with the `addProductHandler()` method, passing the event object.

```
<components:ProductList x="0" y="0" width="100%" height="100%"
    width.cartView="0" height.cartView="0"
    visible.cartView="false"
    dataProvider="{groceryInventory}"
    addProduct="addProductHandler(event)"/>
```

- 8 Next, indicate that you will handle the `ProductList`'s `removeProduct` event with the `removeProductHandler()` method, passing the event object.

```
<components:ProductList x="0" y="0" width="100%" height="100%"
    width.cartView="0" height.cartView="0"
    visible.cartView="false"
    dataProvider="{groceryInventory}"
    addProduct="addProductHandler(event)"
    removeProduct="removeProductHandler(event)"/>
```

- 9 Save this class and run the `FlexGrocer` application.

You should be able to add and remove products again using the buttons, but this action is now performed with events across components in a loosely coupled way.

Handling the Collection Change Event

As you already know, many Flex components and classes, some visual and some non-visual, dispatch events that can be used in your application. In this exercise, you will perform a minor refactoring of the `ShoppingCart` class and use one of these events to ensure that the total of your `ShoppingCart` class always remains correct as you add and remove items.

1 Open `ShoppingView.mxml` from the `views` package.

2 Find the `Label` tag that displays the text `Your Cart Total : $0`.

You will now change this `Label` to reflect the cart's actual total.

3 Change the `Label` to append the `total` property of the `ShoppingCart` instance, named `shoppingCart`, directly after the currency symbol. Surround the expression that retrieves the total in curly brackets, indicating that it should be refreshed if the total changes. Your code should look like this:

```
<s:Label text="Your Cart Total: ${shoppingCart.total}"/>
```

Flex will concatenate the initial portion of that string and the `total` property each time a change in the total is noted. However, there is still one bug in our `ShoppingCart` class that needs to be fixed.

In Lesson 8, “Using DataBinding and Collections,” you added an implicit getter and setter to the `ShoppingCartItem`. Each time the `ShoppingCartItem`'s quantity changes, you update the subtotal for that particular item. Unfortunately, the `ShoppingCart` itself also has a `total` property. Right now, even though the subtotal for each item adjusts correctly, the `ShoppingCart`'s overall total is not aware of that change and will therefore not rerun the `calculateTotal()` method. Effectively, this means that if you update quantities of given items through a method other than `add` or `remove`, the `ShoppingCart` total will not track correctly.

4 Open the `ShoppingCart` class from the `cart` package.

5 As the last item in the class, add a new private method named `handleItemsChanged()`. This method will accept a single parameter named `event` of type `CollectionEvent`.

If you used code-completion, `CollectionEvent` will be imported for you. If not, import `mx.events.CollectionEvent` now. `CollectionEvent` is a special type of event broadcast from collections such as the `ArrayCollection`. It indicates that one of the items in the collection has changed.

6 Inside the `handleItemsChanged()` method, call the `calculateTotal()` method of this object.

```
private function handleItemsChanged( event:CollectionEvent ):void {  
    calculateTotal();  
}
```

Every time the items in the ShoppingCart change, you will respond by recalculating the total for the cart. In this way you can keep track of the changes to the total correctly.

- 7 Find the constructor for the ShoppingCart class. As the last line of the constructor, you'll add an event listener to the `items` ArrayCollection for the `CollectionEvent.COLLECTION_CHANGE` event type. When this event occurs you want the `handleItemsChanged` method called.

```
items.addEventListener(CollectionEvent.COLLECTION_CHANGE, handleItemsChanged );
```

If you use code-completion, Flash Builder will write much of this line on your behalf. This is simply the ActionScript equivalent of adding an event listener in MXML and passing the event object.

The first parameter of the `addEventListener()` call is always a String specifying the type of event. Unfortunately, in ActionScript, unlike in MXML, Flash Builder doesn't look at the event metadata and fill in String on our behalf. It is therefore a common convention to create constants in the system, which are just strings with the name of the event preset on your behalf. This simply prevents you from making a typo by ensuring that the event type you want to listen for does in fact exist.

Last thing to note: When you add an event listener in ActionScript, the second argument is a function reference. So, you don't type `handleItemsChanged(event)` as you would in MXML, but rather just `handleItemsChanged`.

- ▶ **TIP:** If you want to see how the constant works for yourself, hold down the Ctrl (Command) key and click `COLLECTION_CHANGE`. Flash Builder will take you to the `CollectionEvent` class, and you will see a constant. This line of code works the same whether you use the constant or type the string `collectionChange`.

- 8 Find the `addItem()` method and remove the call to `calculateTotal()`.

Any change to the `items` ArrayCollection will now inform the ShoppingCart to recalculate itself. You no longer need to call this explicitly when adding or removing an item.

- 9 Find the `removeItem()` method and also remove the call to `calculateTotal()`.

- 10 Save this class and run the FlexGrocer application.

You can now add and remove items from the cart. As these items change, the total updates automatically as it responds to a notification from the `items` ArrayCollection.

What You Have Learned

In this lesson, you have:

- Learned the benefits of loosely coupled architecture (pages 262–263)
- Dispatched events (pages 263–267)
- Declared events for a component (pages 267–269)
- Identified the need for your own event classes (pages 269–270)
- Created and used an event subclass (pages 270–274)
- Learned about event bubbling (pages 274–279)
- Created the ProductEvent class (pages 280–284)
- Used ProductEvent to add and remove a product from the cart (pages 284–286)
- Used CollectionEvent to update the cart total (pages 287–288)

LESSON 12

What You Will Learn

In this lesson, you will:

- Define the viewable columns of a DataGrid through GridColumn
- Use a `labelFunction` and an `itemRenderer` to display GridColumn information
- Create an MXML component to be used as an item renderer
- Create an inline custom item editor for a GridColumn
- Raise events from inside an item renderer

Approximate Time

This lesson takes approximately 1 hour to complete.

LESSON 12

Using the Flex DataGrid

In Lesson 10, “Using DataGroups and Lists,” you worked with datasets and some controls that can be used to show data. In this lesson, you will build on that set of base controls and be introduced to the primary MXML component used to display and manipulate large datasets.

In this lesson, you will learn how to use the DataGrid component to display a dataset in an interactive way using rows and columns. Aside from using the DataGrid in its simplest form, you’ll learn how to override the default behavior of a particular column in the DataGrid by implementing a custom item renderer; do a custom sort of the data in a column; and change the editing controls that manage the underlying data.



Product	Quantity	Amount	
 Buffalo	1	\$6.5	<input type="button" value="Remove"/>

The shopping cart displayed in a DataGrid

Introducing DataGrids and Item Renderers

Using a DataGrid as a way to display the data of your application enables your users to interact with the data in a large variety of ways. At the simplest level, the DataGrid organizes the data in a column-by-row format and presents this to the user. From there, the DataGrid can be configured to allow you to modify the data it contains.

In this lesson, you will make modifications to FlexGrocer, in which the DataGrid will give you a view of the cart and the ability to update and remove items from the cart.

► **TIP:** Although the DataGrid does provide the most versatile manner of interacting with the data of your application, it also comes with additional overhead (performance and size). It is wise to consider what you expect the user to do with the data or controls before you automatically choose to use a DataGrid.

Displaying the ShoppingCart with a DataGrid

When you finished Lesson 11, “Creating and Dispatching Events,” you had the contents of your cart displayed in a List control with the ability to remove the current item you were viewing via a Remove from Cart button. You will now use a DataGrid to display the contents of the cart. The DataGrid control supports the syntax that allows you to specify the columns explicitly through the GridColumn:

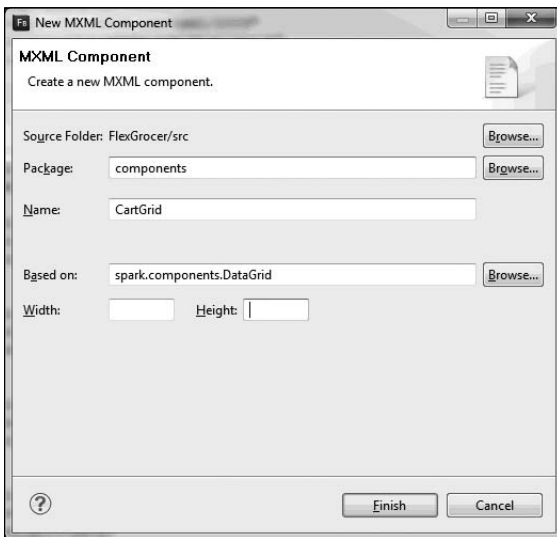
```
<s:DataGrid ... >
  <s:columns>
    <s:ArrayList>
      <s:GridColumn dataField="" .../>
      <s:GridColumn .../>
      <s:GridColumn .../>
    </s:ArrayList>
  </s:columns>
</s:DataGrid>
```

The `dataField` is used to map a property in the dataset to a given column. The order in which the GridColumns are listed is the order in which you will see the columns from left to right in the DataGrid. Unlike earlier versions of Flex, the spark DataGrid requires that you declare a GridColumn instance for every column you want to show. If the DataGrid is declared without any columns, no data will be rendered. Each GridColumn supports a large number of attributes that affect the DataGrid’s rendering and interaction with the given column.

- 1 Locate the components package that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson12/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Right-click the components package and choose New > MXML Component. In the dialog box, specify the Name as **CartGrid**.
- 3 For the "Based on" value, click the Browse button. In the dialog box, begin to type **DataGrid** until you see *DataGrid – spark.components* displayed. Choose the DataGrid entry, click OK, remove the height and width, and then click Finish.



- 4 In the newly created component's `<s:DataGrid>` tag, add the `editable` property and assign it the value `true`.

You are specifying `editable` as `true` because you will allow one of the columns to be changed by the user. If it is set to `false`, the whole DataGrid becomes read-only.

```
<s:DataGrid xmlns:fx="http://ns.adobe.com/mxml/2009"
            xmlns:s="library://ns.adobe.com/flex/spark"
            editable="true">
```

- 5 After the `<fx:Declarations>` tag set, define an `<s:columns>` tag set.

You will be adding `GridColumn` objects in the next steps, and they need to be nested in the `columns` tag set.

- 6** In the `<s:columns>` tag set, define an `<s:ArrayList>` to contain the `ArrayList` of columns. Inside the `ArrayList`, you can begin to define the columns for the grid by adding a `<s:GridColumn>` for the product name. Set the `headerText` to `Product`, `dataField` to `product`, and `editable` to `false`.

```
<s:columns>
  <s:ArrayList>
    <s:GridColumn headerText="Product" dataField="product" editable="false"/>
  </s:ArrayList>
</s:columns>
```

The `headerText` attribute specifies the text of the `GridColumn` header. If you don't specify this, it will take the value of the `dataField` attribute.

Because the `editable` attribute is set to `true` on the `<s>DataGrid>` tag, you need to set it to `false` for each column you don't want the user to be capable of changing.

- 7** Define an `<s:GridColumn>` for displaying the quantity, and place it after the product `<s:GridColumn>`. Set `headerText` to `Quantity` and `dataField` to `quantity`.

```
<s:ArrayList>
  <s:GridColumn headerText="Product" dataField="product" editable="false"/>
  <s:GridColumn headerText="Quantity" dataField="quantity" />
</s:ArrayList>
```

You are not explicitly setting the `editable` property of the quantity `GridColumn` to `false`, as this column will be used to allow users to change the quantity of a specific product they want to buy.

- 8** Define an `<s:GridColumn>` for displaying subtotals for each item and place it after the quantity column. Set `headerText` to `Amount`, `dataField` to `subtotal`, and `editable` to `false`.

```
<s:GridColumn headerText="Amount" dataField="subtotal" editable="false"/>
```

- 9** As a final column, define an `<s:GridColumn>` for displaying a Remove button. At this point, only set `editable` to `false`.

```
<s:GridColumn editable="false" />
```

Later you will add functionality so a button will remove the item in a particular `DataGrid` row.

There is no `dataField`, as you are not expecting the grid to render any property from the dataset in this column. Instead, you'll later add a remove button in this column. Of course, `editable` is `false`, as there is no data to be edited here.

10 At this point, your component should appear as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<s:DataGrid xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  editable="true">
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
  <s:columns>
    <s:ArrayList>
      <s:GridColumn headerText="Product" dataField="product"
        editable="false"/>
      <s:GridColumn headerText="Quantity" dataField="quantity" />
      <s:GridColumn headerText="Amount" dataField="subtotal"
        editable="false"/>
      <s:GridColumn editable="false"/>
    </s:ArrayList>
  </s:columns>
</s:DataGrid>
```

11 Save CartGrid.mxml.

Using the CartGrid Component

You've created the basic component that uses a DataGrid to display data in the shopping cart data structure. Now you will replace the placeholder DataGrid that was inserted earlier with the newly created component.

- 1** Open ShoppingView.mxml from the views package.
- 2** Locate the `<s:DataGrid>` block near the bottom of the file and remove it.
- 3** In the same location, add the `<components:CartGrid>` component. Set the `id` to `dgCart`, the `includeIn` to `cartView`, and the `width` and `height` to `100%`.
- 4** In the `CartGrid`, bind the `dataProvider` to `shoppingCart.items`.

```
<components:CartGrid id="dgCart"
  includeIn="cartView"
  width="100%" height="100%"
  dataProvider="{shoppingCart.items}" />
```

- 5** Run the `FlexGrocer.mxml`. Add products to the shopping cart, and then click the View Cart button.

You should see the DataGrid with some data in it. At this point the information is not formatted correctly, but you see that the component is being used and data is being passed to `CartGrid`. Note the Product column is showing up as text in the DataGrid, even though it is a

complex attribute in the dataset. This is because there is a `toString()` function declared on the `Product` value object. If this weren't defined, you would see `[Object Product]`. You will look at how to better display a complex object later. For now, this is what you should see:

Product	Quantity	Amount
[Product]Bananas	1	1.98
[Product]Buffalo	1	6.5

In this case, the `DataGrid` is set to occupy 100% of the available width. If you wanted the `DataGrid` to size itself to be large enough to fit the items placed in it, you could specify a *typicalItem* for the `dataGrid`. A *typicalItem* is an object that will be used to size the `dataGrid` and its columns. Frequently, these are used to specify the largest possible size a value might have for a `DataGrid` column, so that Flex can ensure that the column is sized appropriately to fit that item. Typical items can be provided through the `typicalItem` property of the `DataGrid`, with this syntax:

```
<s:DataGrid ... >

    <s:typicalItem>
        <fx:Object product="This is a long product name"
            quantity="100"
            subtotal="$99.99"/>
    </s:typicalItem>
    <s:columns>
        <s:ArrayList>
            <s:GridColumn headerText="Product" dataField="product"
                editable="false"/>
            <s:GridColumn headerText="Quantity" dataField="quantity" />
            <s:GridColumn headerText="Amount" dataField="subtotal"
                editable="false"/>
            <s:GridColumn editable="false" />
        </s:ArrayList>
    </s:columns>
</s:DataGrid>
```

Adding Inline Editing Control for GridColumn

Item editors and item renderers can be applied inline, as child tags of a `GridColumn`, or they can be written into a separate class, like the item renderers you saw in Lesson 10. In this first exercise, you will write an inline item renderer.

In a DataGrid, you can specify that a column of the data shown can be changed by the user when focus is brought to the cell. This is done by setting the `editable` attribute to `true`. The default editing control for the column is a text field. It is possible to specify which editor to use when managing the data via the `itemEditor` attribute. By default, an editable column will use a `TextInput` as its item editor, however, you can create your own classes that implement the `IGridItemRenderer` interface, or extend the `GridItemEditor` base class to create any Item editor that you require.

To apply an item editor inline, you will create a child tag of the editable `GridColumn` instance to address the column's `itemEditor` property, and create a new component inside of that. To define a new component, you need to insert an `<fx:Component>` tag, and inside of that define your new item editor.

The syntax looks like this:

```
<s:DataGrid id="dg"
  dataProvider="{myDP}"
  editable="true">
  <s:columns>
    <s:ArrayList>
      <s:GridColumn headerText="Quantity"
        dataField="quantity">
        <s:itemEditor>
          <fx:Component>
            <s:GridItemEditor >
              <s:NumericStepper
                minimum="1"
                maximum="5" />
            </s:GridItemEditor>
          </fx:Component>
        </s:itemEditor>
      </s:GridColumn>
    </s:ArrayList>
  </s:columns>
</s:DataGrid>
```

What this does is define an inner class or a class-within-a-class, which behaves as if everything inside the `<fx:Component>` tag were defined in a different class file. This inner class can contain any elements that an MXML class might contain, such as script, components, and so on. Keep in mind that within this inner class, the keyword `this` now refers to is the inner class, not to the class defined as the root of that file. If you need to reference elements from the parent class while in the inner class, you can use the keyword `outerDocument`.

When a user double-clicks a cell in an editable column, the `itemRenderer` is replaced with an `itemEditor`. The value from the `dataField` for that column is passed to the setter for the `itemEditor`'s `value` property. When the user is finished editing the data and clicks away from

the cell, the DataGrid will read from the getter for the itemEditor's value property and set that value back into the underlying data.

As you create your own itemEditors, you will want to provide your own getter and setter for the value property.

- 1 Open the CartGrid.mxml file that you created in a previous exercise.
- 2 In the `<s:GridColumn>` tag that maps to the quantity, change the self-closing tag to a tag pair. Add a child tag that addresses the `itemEditor` property.

```
<s:GridColumn headerText="Quantity" dataField="quantity">
  <s:itemEditor>
</s:itemEditor>
</s:GridColumn>
```

Next, you will create the inner class that has the specifics for your itemEditor.

- 3 Inside the `itemEditor` tag pair, add an `<fx:Component>` tag pair. As a child to the `<fx:Component>` tag, create an instance of the `GridItemEditor` class.

```
<s:GridColumn headerText="Quantity" dataField="quantity">
  <s:itemEditor>
    <fx:Component>
      <s:GridItemEditor >

      </s:GridItemEditor>
    </fx:Component>
  </s:itemEditor>
</s:GridColumn>
```

`GridItemEditor` is now the super class for your new inner class.

- 4 Add a `NumericStepper` as a child to the `GridItemEditor`. Give the `NumericStepper` an instance name of `qty`.

```
<s:GridColumn headerText="Quantity" dataField="quantity">
  <s:itemEditor>
    <fx:Component>
      <s:GridItemEditor >
        <s:NumericStepper id="qty" />
      </s:GridItemEditor>
    </fx:Component>
  </s:itemEditor>
</s:GridColumn>
```

Now your grid can display a `NumericStepper` as the editor. But, if you try to edit the value, the `NumericStepper` won't show the correct value, nor will the proper data be stored back in the `dataProvider` when you are finished editing. You will fix this in the next step.

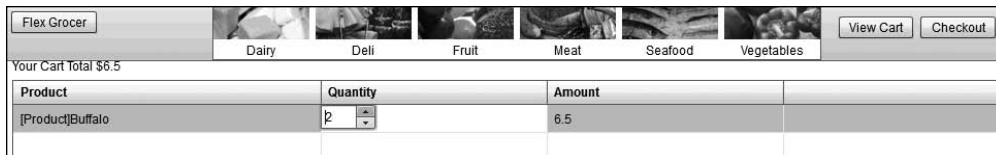
- 5 After the opening GridItemEditor tag, but before the NumericStepper, add an `<fx:Script>` block. In the block, override the `get value` method and return `qty.value`. Also override the `set value` method and set the incoming parameter as the value of `qty.value`. You will need to cast the incoming parameter to an integer.

```
<fx:Script>
<![CDATA[
    override public function get value():Object{
        return qty.value;
    }

    override public function set value(newValue:Object):void{
        qty.value = int(newValue) ;
    }
]]>
</fx:Script>
```

- 6 Save `CartGrid.mxml`. Run the `FlexGrocer` application, add the `Buffalo` product to the shopping cart, and click `View Cart`.

When you double-click in the `Quantity` column, you will notice that it doesn't open as a free-form text field, but rather as a `NumericStepper` control.



Product	Quantity	Amount
[Product]Buffalo	2	6.5

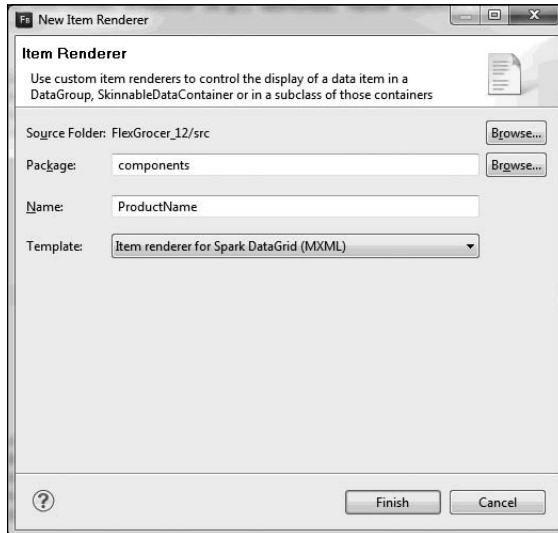
Creating an Item Renderer for Displaying the Product

The default behavior of the `DataGrid` is to convert every value of the dataset to a string and then display it. However, when you are dealing with a complex object that is stored in the dataset, another alternative is to create a custom item renderer that shows more information about the column. In this case, you're going to create a simple item renderer that shows the product's name and image.

When working with item renderers, you'll find that there is an implicit public variable available to you in the item renderer called `data`, which represents the data of the row itself. You can use `data` to bind your controls without having to worry about which column you are working with. When the `DataGrid` creates a column that has a custom item renderer associated with it, it creates a single instance of the cell renderer per row, so you don't have to worry about which row of data you are working with in a renderer.

ItemRenderers for a GridColumn also have a public method called `prepare`. This method is called after the data has been set for the cell, but before the cell is rendered. Rather than use data binding to populate the controls, it can be more efficient to populate the controls in the `prepare` method.

- 1 Right-click the components package and choose `New > Item Renderer`. In the New Item Renderer dialog box, verify the Package is set to **components**, then set the Name to **ProductName**, and choose **Item Renderer for Spark DataGrid (MXML)** as the Template.



This MXML file will define the layout of a given cell in the DataGrid. You are creating it in a separate file so that, if needed, it can be used on multiple DataGrid columns and/or multiple DataGrids.

- 2 Examine the code that was created for you:

```
<?xml version="1.0" encoding="utf-8"?>
<s:GridItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  clipAndEnableScrolling="true">

  <fx:Script>
    <![CDATA[
      override public function prepare(
        ↪ hasBeenRecycled:Boolean):void {
          lblData.text = data[column.dataField]
        }
    ]]>
  </fx:Script>
</s:GridItemRenderer>
```

```

    ]]>
</fx:Script>

<s:Label id="lblData" top="9" left="7"/>

</s:GridItemRenderer>

```

Notice that there is a root node of `GridItemRenderer`, which has its `clipAndEnableScrolling` property set to `true`. This ensures that none of the contents of the item renderer extend past the boundaries of the renderer. Also notice that the `Script` block contains an override of the `prepare` method. In this method, the `text` property of the label is set to show the value evaluated from the `column.dataField` of the data.

- 3 Add a `VerticalLayout` as the value for the `layout` property of the `GridItemRenderer`.

```

<?xml version="1.0" encoding="utf-8"?>
<s:GridItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  clipAndEnableScrolling="true">
  <s:layout>
    <s:VerticalLayout />
  </s:layout>

```

In this renderer, you will have two children (an image and a label) that you want to lay out vertically.

- 4 Place an `<s:Image>` tag after the end of the `Script` block. Give the Image an `id` of `prodImage`, and a width of `100`.

This image will be used to show a thumbnail of the product.

- 5 Remove the code inside the `prepare` method. In its place, check if the data property has a value. If it does, create a local variable named `prod`, with a data type of `Product` that gets its value from `data.product`. Assign `lblData.text` to show `prod.prodName`, and assign the source of `prodImage` to be `"assets/"` concatenated with `prod.imageName`. If data has no value, set the text of `lblData` to an empty string, and the source of `prodImage` to `null`.

```

override public function prepare(hasBeenRecycled:Boolean):void {
  if(data){
    var prod:Product = data.product as Product;
    lblData.text = prod.prodName;
    prodImage.source = "assets/"+prod.imageName;
  } else {
    lblData.text = "";
    prodImage.source = null;
  }
}

```

If you use code-completion, `valueObjects.Product` will be imported for you. If not, be sure to import it manually.

- ▶ **TIP:** The image location used is relative to the location from which the main application was loaded, not the location of the file that contains the `<s:Image>` tag.

This will now populate both the label and the image with the proper elements from the provided data.

- 6 Save the `ProductName.mxml` file.

You cannot test this component at this time because it is not yet assigned to the `DataGrid`.

- 7 Open the `CartGrid.mxml` you created in the previous exercise. Add a `variableRowHeight` attribute to the root node, with the value `true`.

It is necessary for you to set the `variableRowHeight` to `true` so that Flex resizes the row's height to accommodate the thumbnail image.

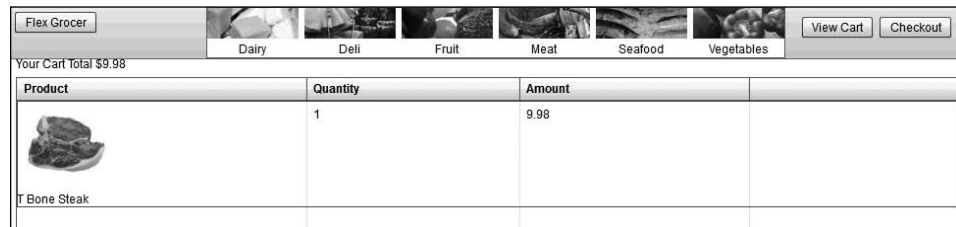
- ▶ **TIP:** This attribute can be used to allow for exploding details inside a `DataGrid` row. In this case, you can have summary data in a cell that expands to show details if you click an icon or button.

- 8 Update the `<s:GridColumn>` with a `dataField` of `product` with a new attribute, `itemRenderer`, set to `components.ProductName`.


```
<s:GridColumn headerText="Product"
  dataField="product"
  editable="false"
  itemRenderer="components.ProductName" />
```

You need to use the fully qualified class name to set your item renderer for a `GridColumn`.

- 9 Save `CartGrid.mxml`. Run the `FlexGrocer` application, add a product to the shopping cart, and click `View Cart`.



The screenshot shows the FlexGrocer application interface. At the top, there is a navigation bar with a "Flex Grocer" button and several category buttons: Dairy, Deli, Fruit, Meat, Seafood, and Vegetables. On the right side of the navigation bar, there are "View Cart" and "Checkout" buttons. Below the navigation bar, it says "Your Cart Total \$9.98". The main content area displays a DataGrid with the following structure:

Product	Quantity	Amount
 T Bone Steak	1	9.98

Creating an Inline MXML Item Renderer for Displaying a Remove Button

In the same way that you created an `itemEditor` inline, you can also create `itemRenderers` inline as well. From a compiler perspective, creating an inline item renderer is the equivalent of building it in an external file (it actually compiles the code of the inline item renderer as a separate file internally). Inside the `<s:itemRenderer>` tag, you will place an `<fx:Component>` tag that defines the boundaries of the inline item renderer file from the rest of the page. Thus, the inside of the `<fx:Component>` tag will have its own scope for which you will need to do imports, function declarations, and the like.

- TIP:** Although building inline item renderers will be very efficient from a coding perspective, you won't be able to reuse the item renderers for other DataGrids. Good candidates for inline renderers or editors are those that are specific to one DataGrid only, such as action item controls.

In this exercise you will add a Remove button to each row.

- 1 Open the `CartGrid.mxml` you created in the previous exercise.
- 2 Locate the fourth `<s:GridColumn>` and change it to a `GridColumn` tag pair by removing the `</>` at the end of the tag and adding just the `>` back on.

```
<s:GridColumn editable="false">
</s:GridColumn>
```

This is the placeholder column in the DataGrid. You'll use `<s:GridColumn>` tag pair because the item renderer definition will be placed inside it. You do not need to specify `dataField`, because there is no data you are mapping directly to.

- 3 Place an `<s:itemRenderer>` tag set and an `<fx:Component>` tag set inside the `<s:GridColumn>` tag.

```
<s:GridColumn editable="false">
  <s:itemRenderer>
    <fx:Component>
    </fx:Component>
  </s:itemRenderer>
</s:GridColumn>
```

- 4 Place an `<s:GridItemRenderer>` tag inside the `<fx:Component>` tags to provide a container for the Remove button.

```
<s:itemRenderer>
  <fx:Component>
    <s:GridItemRenderer>
    </s:GridItemRenderer>
  </fx:Component>
</s:itemRenderer>
```

- Place an `<s:Button>` tag as a child of `GridItemRenderer`. Set the `label` to `Remove` and set the `click` event to call a `removeItem()` function that you will create in the next exercise. Pass data as `ShoppingCartItem` as the parameter to the method. An `import` statement for `valueObjects.ShoppingCartItem` should be added automatically to the inline component. If not, add an `<fx:Script>` block inside the `<s:GridItemRenderer>` tag, and include the `import` statement. Specify a `horizontalCenter` and `verticalCenter` attribute with values of `0`.

```
<s:GridItemRenderer>
  <fx:Script>
    <![CDATA[
      import cart.ShoppingCartItem;
    ]]>
  </fx:Script>
  <s:Button label="Remove"
    click="removeItem(data as ShoppingCartItem)"
    horizontalCenter="0" verticalCenter="0" />
</s:GridItemRenderer>
```

You need to add the appropriate `import` statement, because the `import` statements made at the top of the file are in a scope different from the inline item renderer. The `horizontalCenter` and `verticalCenter` attributes will keep the `Button` centered in the cell.

Reusing the ProductEvent Class

At this point there is no `removeItem()` method in your renderer. When you create this method you will reuse code created in the previous lesson. In Lesson 11, you created an event subclass to hold a `Product` value object. Now you will reuse that event subclass to dispatch the `Product` object you want removed from the shopping cart.

- Inside the `<fx:Script>` block, which is inside the `GridItemRenderer` you created in the last section, create a private function named `removeItem()` that returns `void` and that accepts one parameter, named `item`, of type `ShoppingCartItem`.

```
private function removeItem( item:ShoppingCartItem ):void {
}
```

- Inside the `removeItem()` method, declare a new local variable named `prodEvent` of type `ProductEvent`, and assign it a new instance of the `ProductEvent` event class. For the type parameter of the `ProductEvent` constructor, pass the event name `removeProduct`. Then pass the `item.product` value as the second constructor parameter. Finally, dispatch the event.

```
private function removeItem( item:ShoppingCartItem ):void{
  var prodEvent:ProductEvent = new ProductEvent(
    "removeProduct", item.product);
  dispatchEvent(prodEvent);
}
```

If you use code-completion, `events.ProductEvent` will be imported for you. If not, be sure to import it manually. This event will now be dispatched from the `itemRenderer` and will bubble up toward `ShoppingView`.

- 3 Outside the `GridItemRenderer`, just after the opening `DataGrid` tag, add an `<fx:Metadata>` tag. Inside it, declare that `CartGrid.mxml` will dispatch an event named `removeProduct`. Indicate that the event will be of type `events.ProductEvent`.

```
<fx:Metadata>
    [Event(name="removeProduct", type="events.ProductEvent")]
</fx:Metadata>
```

You are now dispatching the `Product` object you wish to be removed from the shopping cart. Of course, to actually have it removed you must handle the dispatched event, which you will now do in the next steps.

- 4 Save the file.
- 5 Open `ShoppingView.mxml`. Locate the instantiation of `CartGrid` you coded earlier in this lesson.
- 6 Place your cursor in the tag and press `Ctrl-Spacebar` to bring up code completion. Select the `removeProduct` event that you just created in `CartGrid`. For the event handler call the previously created `removeProductHandler()` method and pass the event object as a parameter.

```
<components:CartGrid id="dgCart"
    includeIn="cartView"
    width="100%" height="100%"
    dataProvider="{shoppingCart.items}"
    removeProduct="removeProductHandler( event )"/>
```

The extended event is handled in the `CartGrid` component. The `removeProductHandler()` method you built in the previous lesson performs the removal of the product from the cart.

- 7 Run `FlexGrocer` and add items to the cart. Click the `View Cart` button and confirm that you can remove items from the cart using the `Remove` button.

Creating a labelFunction to Display the Subtotal

You need to create a `labelFunction` to display the subtotal in the third column of the `DataGrid`. Recall that in Lesson 10 you created a `labelFunction` to display the product name in a `List` component. The method signature for a `labelFunction` on a `DataGrid` is `labelFunctionName(item:Object, dataField:GridColumn)`.

- 1 In CartGrid.mxml, create a new `<fx:Script>` block.
- 2 Inside the `<fx:Script>` block, add a private function named `renderPriceLabel` with the arguments `item` typed as a `ShoppingCartItem` and `column` with the datatype `GridColumn`. The function itself will return a `String`.

```
private function renderPriceLabel( item:ShoppingCartItem,
  ➤ column:GridColumn ):String{
}
```

If you use code-completion, `cart.ShoppingCartItem` will be imported for you. If not, be sure to import it manually.

Because the `DataGrid` has multiple columns that can each have its own `labelFunction`, as well as share the same `labelFunction`, the second argument is used to distinguish between which `GridColumn` is using the `labelFunction`. If you know that your function will be used on just one column, you can ignore the second argument in your code.

- 3 As the first line of code in the `renderPriceLabel()` function, create a variable local to the function named `subtotal` with the datatype `Number`, and assign it the particular column's `dataField` value from the `item`.

```
var subtotal:Number = item[ column.dataField ];
```

If you were not creating this function for use by multiple columns, you could have assigned the variable simply as `item.subtotal`. This would have assigned the correct value. But, since you want the function to be reusable, you use the column name to retrieve the correct data, hence `item[column.dataField]`.

- 4 As the last line of code in the function, return the subtotal of the item formatted with a \$.

For now, you want to put a simple mask on the price to represent the number as a dollar figure. The signature and functionality of the `labelFunction` is the same on the `DataGrid` as it is on the `List`.

```
private function renderPriceLabel( item:ShoppingCartItem,
  ➤ column:GridColumn ):String{
    var subtotal:Number = item[ column.dataField];
    return "$" + String( subtotal );
}
```

- 5 Update the `<s:GridColumn>` with a `dataField` of `subtotal` with a new attribute of `labelFunction` set to `renderPriceLabel`.

```
<s:GridColumn dataField="subtotal" headerText="Amount"
  labelFunction="renderPriceLabel" editable="false"/>
```

This will have the `subtotal` column use `renderPriceLabel` on each of the rows in the `DataGrid`.

6 Check the code for the component you have built.

The final code for the CartGrid.mxml should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<s:DataGrid xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  editable="true"
  variableRowHeight="true">
  <fx:Metadata>
    [Event(name="removeProduct",type="events.ProductEvent")]
  </fx:Metadata>
  <fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
  </fx:Declarations>
  <fx:Script>
    <![CDATA[
      import cart.ShoppingCartItem;
      private function renderPriceLabel(
        ✎ item:ShoppingCartItem, column:GridColumn):String{
          var subtotal:Number = item[ column.dataField ];
          return "$" + String( subtotal);
        }
      ]]>
  </fx:Script>

  <s:columns>
    <s:ArrayList>
      <s:GridColumn headerText="Product"
        dataField="product"
        editable="false"
        itemRenderer="components.ProductName"/>
      <s:GridColumn headerText="Quantity" dataField="quantity">
        <s:itemEditor>
          <fx:Component>
            <s:GridItemEditor >
              <fx:Script>
                <![CDATA[
                  override public function get value():Object{
                    return qty.value;
                  }

                  override public function set value(newValue:Object):
                    ✎ void{
                      qty.value = int(newValue) ;
                    }
                ]]>
              </fx:Script>
            <s:NumericStepper id="qty" />
          </fx:Component>
        </s:itemEditor>
      </s:GridColumn>
    </s:ArrayList>
  </s:columns>
</s:DataGrid>
```

```

        </s:GridItemEditor>
    </fx:Component>
</s:itemEditor>
</s:GridColumn>
<s:GridColumn headerText="Amount"
    dataField="subtotal"
    editable="false"
    labelFunction="renderPriceLabel"/>
<s:GridColumn editable="false">
    <s:itemRenderer>
        <fx:Component>
            <s:GridItemRenderer>
                <fx:Script>
                    <![CDATA[
                        import cart.ShoppingCartItem;

                        import events.ProductEvent;

                        private function removeItem(
                            ➤ item:ShoppingCartItem ):void(
                                var prodEvent:ProductEvent = new ProductEvent(
                                    ➤ "removeProduct", item.product );
                                    dispatchEvent(prodEvent);
                                }
                            ]]>
                </fx:Script>
                <s:Button label="Remove"
                    click="removeItem(data as ShoppingCartItem)"
                    horizontalCenter="0" verticalCenter="0" />
            </s:GridItemRenderer>
        </fx:Component>
    </s:itemRenderer>
</s:GridColumn>
</s:ArrayList>
</s:columns>
</s:DataGrid>

```

- 7** Save CartGrid.mxml. Run the FlexGrocer.mxml application, add the Buffalo product to the shopping cart, and click View Cart. Notice both the formatting on the Amount column and the Remove button in the shopping cart.



What You Have Learned

In this lesson, you have:

- Displayed a dataset via a `DataGrid` (pages 292–293)
- Defined the viewable columns of a `DataGrid` through `GridColumn` (pages 293–296)
- Created an `itemEditor` to allow users to change data in a `DataGrid` (pages 296–299)
- Created an `MXML` component to be used as an item renderer (pages 299–303)
- Created an inline custom item renderer for a `GridColumn` (pages 304–305)
- Displayed information from a `GridColumn` using a `labelFunction` and an item renderer (pages 303–308)
- Learned how to raise events from inside an item (pages 305–306)

LESSON 13

What You Will Learn

In this lesson, you will:

- Learn the terminology associated with drag-and-drop operations in Flex
- Understand that the list-based components in Flex have enhanced drag-and-drop support built in
- Implement drag and drop on drag-enabled components
- Use various drag events
- Implement various methods of the DragSource and DragManager classes to implement drag and drop on non-drag-enabled components
- Use formats to allow the dropping of drag proxy objects

Approximate Time

This lesson takes approximately 1 hour and 30 minutes to complete.

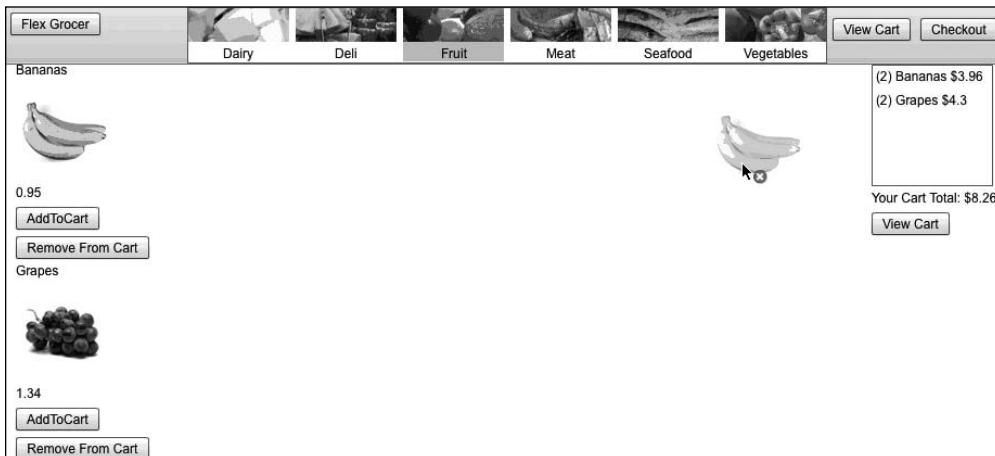
LESSON 13

Using Drag and Drop

Drag and drop is a common user interface technique in desktop applications. It was not so, however, in web applications until the rich Internet applications (RIAs) came along. Flex and Flash Player permit you as a web developer to use drag and drop just as a desktop developer does.

To implement drag and drop in a Flex application, you use the Drag and Drop Manager and the tools it provides. The Drag and Drop Manager enables you to write a Flex application in which users can select an object and then drag it to, and drop it on, a second object. All Flex components support drag-and-drop operations, and a subset has additional drag-and-drop functionality in which implementation requires little more than adding a single property.

In this lesson, you will implement drag and drop in your e-commerce application so a user can click a product and drag it to the shopping cart.



Dragging a grocery item to the shopping cart

Introducing the Drag and Drop Manager

The first step in understanding the Drag and Drop Manager is to learn the terminology surrounding it. The terminology is summarized in the following table.

Drag and Drop Manager Terminology	
Term	Definition
Drag initiator	Component or item from which a component is being dragged.
Drag source	Data being dragged.
Format	Property of the DragSource that provides a description of the format of the data being dragged.
Drag proxy	Image displayed during the dragging process.
Drop target	Component that the drag proxy is over.

The following figure gives you a visual representation of the terminology:

Product	Price	Low Fat	Organic
Fuji Apple	89	Yes	No
Navel Orange	0.43	Yes	No
Romain Lettuce	1.25	Yes	Yes
Spinach	0.49	Yes	No

Navel Orange
Romain Lettuce

Spinach 0.49 Yes No

There are three phases to a drag-and-drop operation:

- 1. Initiating:** A user clicks a Flex component or an item in a Flex component and then begins to move the component or item while holding down the mouse. The component or item is the drag initiator.
- 2. Dragging:** While holding down the mouse button, the user moves the mouse around the screen. Flex displays an image called a drag proxy, and the associated non-visual object called the drag source holds the data associated with the component or item being dragged.
- 3. Dropping:** When the user moves the pointer over another component that will allow it, the item can be dropped on a drop target. The data is then inserted into the new component in some way.

Flex components fall into two groups when it comes to drag-and-drop support: those with enhanced drag-and-drop functionality and those without. In the current release of Flex, the only component with this enhanced functionality is the List control. You can expect that to change in future minor and major releases.

What this means to you as a developer is that your life will be a little bit easier when implementing drag and drop with those controls that have enhanced support. In fact, in many cases implementing might require no more than setting a single property value for each of the controls involved in the drag-and-drop operation.

Enhanced Dragging and Dropping Between Two Lists

Your first foray into implementing drag-and-drop operations in Flex will be between two List instances. Because they are list-based components and have enhanced drag-and-drop support, you will need to write very little code.

Two properties are important in this first phase: `dragEnabled` and `dropEnabled`. Here are their descriptions:

- `dragEnabled`: Assigned a Boolean value to specify whether the control is allowed to act as a drag initiator (defaults to `false`). When it's `true`, the user can drag items from the component.
- `dropEnabled`: Assigned a Boolean value to specify whether the control is allowed to act as a drop target (defaults to `false`). When it's `true`, the user can drop items onto the control using the default drop behavior.

To enable dragging and dropping, simply set the `dragEnabled` property in the component from which you are dragging to `true` and set the `dropEnabled` property in the component on which you are dropping to `true`.

Now you will put your drag-and-drop knowledge to use by implementing drag and drop from one List to another List.

- 1** Import the `DragDropStart.fxp` from the `Lesson13/independent` folder into Flash Builder. Please refer to the appendix for complete instructions on importing a project.
- 2** Open the `Task1_List_to_List.mxml` file.

You will use this project instead of the `FlexGrocer` one because some of the work in this lesson will not be directly involved with the `FlexGrocer` site.

- 3** Examine the code in the `Task1_List_to_List.mxml` file and then run it.

Note that the existing code does not use any concepts you have not already learned in this book. The file uses an `HTTPService` remote procedure call (RPC) to retrieve grocery info. The file then uses a `result` handler to place the data into an `ArrayCollection`, which is then used as a `dataProvider` in a `List`. When you run the application, you see you have a `List` populated with grocery product information and another `List` below it. When you try to drag and drop between the `List` instances, you'll see that this functionality is not yet working.

- 4** In the first `List`, set the `dragEnabled` property to `true`. Run the application; you can click one of the rows in the `List` and drag the drag proxy around the screen.

Setting this property did two obvious things: It enabled dragging and created the drag proxy, the image attached to the pointer when dragging. Another non-visual event occurred at the same time: A `DragSource` object was created to hold the data.

A `DragSource` is just an object that holds the data being dragged and associates it with a `String` called a `format`. By default, the data is associated with a `format` named `itemsByIndex`, as the following figure from the debugger shows:

Name	Value
▶ this	Task1_List_to_List (@10b5c0a1)
▶ <scope chain>	
▲ event	mx.events.DragEvent (@cb684a1)
▶ [inherited]	
action	"copy"
draggedItem	null
dragInitiator	spark.components.List (@185210a1)
▲ dragSource	mx.core.DragSource (@117dbf01)
dataHolder	Object (@11599e09)
formatHandlers	Object (@11599e51)
▲ formats	[] (@117e8ab1)
[0]	"itemsByIndex"
[1]	"caretIndex"
length	2
formats	[] (@117e8ab1)

- 5** In the second `List`, set the `dropEnabled` property to `true`. Your `Lists` should appear as follows:

```
<s:List id="initiatorList"
  dataProvider="{products}"
  labelField="name"
  dragEnabled="true"/>
```

```
<s:List id="targetList"
  labelField="listPrice"
  dropEnabled="true"/>
```

You've done two basic steps so far to enable drag-and-drop for the application:

- Added the `dragEnabled` property to the drag initiator
- Added the `dropEnabled` property to the drop target

Now you're ready to test.

- 6 Run the application and drag from the first List and drop onto the second.

When you drag an item from the first List to the second, you are not dragging the name of the food. You are dragging the data associated with that product. So, even though the `listPrice` property was not displayed in the first List, it is available when you drop it in the second. This shows you that all the data for the object being dragged is in the `DragSource`, not just the properties that happen to be displayed.

Standard Dragging and Dropping Between a DataGrid and a List

In the description of the `dropEnabled` property, the following sentence was used: "When it's true, the user can drop items onto the control using the default drop behavior." So what is this "default drop behavior"? Basically it means that Flex will add a reference to the item in the original List to the `dataProvider` of the new List. In other words, both Lists will now show the same items. However, you might want a different outcome. Perhaps you want to move, instead of copy, the items between Lists. In this task you'll drag from a List to a DataGrid component. In this case, the Spark DataGrid does not support "default drop behavior" so it won't know how to handle the data dragged from the List component.

You'll use the events broadcast by Flex controls to handle the dropping onto the DataGrid. Here's a summary of the events for both the drag initiator and the drop target:

Drag Initiator Events	
Drag Events	Description
<code>mouseDown</code> and <code>mouseMove</code> (<code>MouseEvent</code> class)	Not drag events but used to start the drag-and-drop process when not using <code>dragEnabled</code> components. The <code>mouseDown</code> event is broadcast when the user selects a control with the mouse and holds down the mouse button. The <code>mouseMove</code> event is broadcast when the mouse moves.
<code>dragComplete</code> event (<code>DragEvent</code> class)	Broadcast when a drag operation is completed, either when the drag data drops onto a drop target or when the drag-and-drop operation ends without performing a drop operation.

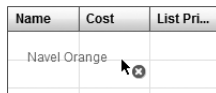
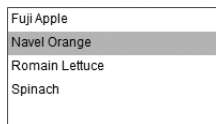
Drop Target Events

Drag Events (all events of the DragEvent class)	Description
dragEnter	Broadcast when a drag proxy moves over the target from outside the target.
dragOver	Broadcast as the user moves the pointer over the target, after the dragEnter event.
dragDrop	Broadcast when the mouse is released over the target.
dragExit	Broadcast when the user drags outside the drop target, but does not drop the data onto the target.

Now it is time to get to work.

- 1 Examine the code in the Task2_List_to_DG.xml file and then run it. Try to drag from the List to the DataGrid; you'll see that you can drag an item from the List but cannot drop it on the DataGrid and that the cursor will change to a red circle with an X indicating you are not allowed to drop this data on the DataGrid.

Unlike the previous lesson, you cannot simply add the `dropEnabled` property to the DataGrid. That property does not exist. Instead you must handle some of the events by yourself.



- 2 Begin adding a `dragEnter` event listener to the DataGrid, and select Generate DragEnter handler to have Flash Builder create the event handler for you. The generated code calls the newly created event handler and passes the event object as a parameter.

```
<s:DataGrid id="targetGrid"
    width="200"
    dragEnter="targetGrid_dragEnterHandler(event)"
    dataProvider="{targetListDP}">
<s:columns>
    <s:ArrayList>
        <s:GridColumn headerText="Name"
```

```

        dataField="name"/>
    <s:GridColumn headerText="Cost"
        dataField="cost"/>
    <s:GridColumn headerText="List Price"
        dataField="listPrice"/>
</s:ArrayList>
</s:columns>
</s:DataGrid>

```

- 3** Check that the event handler was created in the `<fx:Script>` block.

Note that the event is data typed as `DragEvent` and the function will not return any data, so the data type is `void`.

The `dragEnter` event is broadcast when you drag an item from the List over the DataGrid. Since a user could drag any number of things over a control in Flex, the `dragEnter` handler offers you an opportunity to inspect the data being dragged and determine if you want to allow it to be dropped on the DataGrid.

- 4** Remove the TODO comment from the newly created event handler. As the first line of code in the function, create a new local variable named `dragSource` of type `DragSource` and set it equal to the `dragSource` property of the event:

```

protected function targetGrid_dragEnterHandler(event:DragEvent):void {
    var dragSource:DragSource = event.dragSource;
}

```

The `DragEvent` object has a property named `dragSource` that is an instance of the `DragSource` class. This property contains information about the items being dragged. Remember to import the `DragSource` class.

- 5** Just below this line, add an `if` statement that checks the result of calling the `dragSource.hasFormat()` method and passing it the `itemsByIndex` String:

```

protected function targetGrid_dragEnterHandler(event:DragEvent):void {
    var dragSource:DragSource = event.dragSource;

    if ( dragSource.hasFormat( "itemsByIndex" ) ) {
    }
}

```

The `hasFormat()` method checks to see if that data being dragged has a certain format. For example, here you are checking to see if the data just dragged over the DataGrid has a format named `itemsByIndex`. In general, formats are used to determine if an application or control in the application can display certain data. For example, the DataGrid is good at displaying rows of data whereas an image control is useful at displaying bitmap data. So, if the format dragged over the DataGrid were rows of data then you might tell the

user it is okay to drop that data here. However, if the format were a bitmap or a PDF, then you might not want to let the user drop that data on this control. In all cases, the control that will receive the data gets to make that decision.

*** NOTE:** Remember that the default format name associated with data in a List is always `itemsByIndex`.

- 6** Inside the `if` statement, you will call the static `acceptDragDrop()` method of the `DragManager` class and pass it the `event.target` cast as a `IUIComponent`:

```
protected function targetGrid_dragEnterHandler(event:DragEvent):void {
    var dragSource:DragSource = event.dragSource;

    if ( dragSource.hasFormat( "itemsByIndex" ) ) {
        DragManager.acceptDragDrop( event.target as IUIComponent );
    }
}
```

This code reads: When the user moves a dragged item over the `DataGrid`, check to see if it contains a bunch of items in order of their index. If it does, then tell the `DragManager` that this component (the `DataGrid` in this case) is willing and able to accept these items should the user now let go of the mouse button.

- 7** Save and run the application.

This time when you drag items from the List to the `Datagrid`, the cursor will change from a red circle to the regular cursor, indicating that you may drop these items on the `DataGrid`. If you drop the items at this moment, you'll notice that nothing happens. That will be handled next.

- 8** Add a `dragDrop` event listener to the `DataGrid`, and select `Generate DragDrop` handler to have Flash Builder create the event handler for you. The code generated calls the newly created event handler and passes the event object as a parameter.

The `dragDrop` event is broadcast when you let go out of the mouse button while dragging an item from the List over the `DataGrid`. The `dragDrop` handler allows you to manipulate or use the data being dragged and determine what to do with it.

- 9** Check that the `targetList_dragDropHandler` event handler was created in the `<fx:Script>` block. Inside of this method, add a local variable named `items` of type `Vector.<Object>`:

```
var items:Vector.<Object>;
```

This syntax may seem extremely strange. `Vector` is a type of `Object` you have not yet dealt with in `ActionScript`. Put simply, `Vectors` are typed arrays. Meaning that they are arrays

that can contain only one type of Object at a time. Normal ActionScript arrays can contain anything, meaning that this is completely valid:

```
var ar:Array = new Array();
ar[ 0 ] = 1;
ar[ 1 ] = "Hello";
ar[ 2 ] = new Button();
```

Vectors are different. When you create a Vector, you explain what type of Object will be contained within the Vector. Trying to insert any other type of Object throws an error. This allows the Vector to be much quicker and more efficient than the standard Array. When defining a Vector, you use the following syntax:

```
var name:Vector.<Type>;
```

For example:

```
var myVector:Vector.<int>;
var yourVar:Vector.<Button>;
```

The enhanced drag and drop API uses Vectors to represent all dragged rows, therefore you will need to use Vectors to access this data.

- 10** On the next line, in the method, create a new local variable named `dragSource` of type `DragSource` and set it equal to the `dragSource` property of the event:

```
protected function targetGrid_dragDropHandler(event:DragEvent):void {
    var items:Vector.<Object>;
    var dragSource:DragSource = event.dragSource;
}
```

- 11** Call the `dataForFormat()` method on the `dragSource` and pass it `itemsByIndex`. This method returns the data associated with the `itemsByIndex` format. Because you are using the enhanced List drag and drop support, the data will be a Vector containing objects, so cast the result as a `Vector.<Object>` and assign it to the `items` variable:

```
protected function targetGrid_dragDropHandler(event:DragEvent):void {
    var items:Vector.<Object>;
    var dragSource:DragSource = event.dragSource;

    items = dragSource.dataForFormat( "itemsByIndex" ) as Vector.<Object>;
}
```

- 12** Set a breakpoint at the closing brace of the `targetList_dragDropHandler(event:DragDropEvent)` function. You do this by double-clicking in the marker bar just to the left of the line numbers in the editor. You will see a small blue dot appear to indicate that the breakpoint was set.

The breakpoint will cause Flash Builder to halt execution at the marked line of code, and you'll be able to check values of variables. Recall that you first learned about debugging in Lesson 2, "Getting Started."

- 13 Debug the application and drag a row to the DataGrid. When you drop the drag proxy, the process flow will return to Flash Builder. Open the Flash Debug perspective. Examine the `items` variable value in the Variables view. You should see that the variable contains a Vector containing the data from that List.

The following figure shows the row of data being dragged:

Name	Value
items	__AS3__vec.Vector.<Object> (@a150c11)
[inherited]	
[0]	mx.util.ObjectProxy (@a1c8a81)
category	"Fruit"
cost	"0.43"
description	null
dispatcher	flash.events.EventDispatcher (@a26d831)
id	null
imageName	"navel.jpg"
isLowFat	"Yes"
isOrganic	"No"
item	Object (@a0e9611)
listPrice	"0.99"
name	"Navel Orange"
notifiers	Object (@a0e95e1)
object	Object (@a0e9611)
productId	3
propertyList	null
proxyClass	Object (@92cd7c1)
proxyLevel	-1 [0xffffffff]
toDisplay	false
type	null
_type	null
uid	"2621C840-D13D-4902-0FAF-BCE1228DE58C"
unit	"Each"
dragSource	mx.core.DragSource (@a4432e1)

Notice that the variable contains an array of length 1, which means you have only 1 index, which is 0.

- ▶ **TIP:** If you want to allow the user to drag multiple rows of data, set the `List allowMultipleSelection` property equal to `true`.

- 14 Terminate the debugging session by clicking the red square in either the Debug or Console views. Return to the Flash perspective by clicking the chevron (>>) in the upper-right corner of Flash Builder and selecting that perspective.

Normally, the Flash perspective is best to work in because you can see much more of your code.

- 15** As the next line in the function, add the dragged product to the DataGrid by using the `addItem()` method of the DataGrid's `dataProvider`. Remember that the `items` variable contained an array of length 1, so use `items[0].name` to reference the name.

```
targetListDP.addItem( items[ 0 ] );
```

This is a case in which viewing how the data is stored using the debugger is very helpful in retrieving the information.

- 16** Run the application and drag from the List to the DataGrid. You should see the product being placed in the DataGrid.

Fuji Apple
Navel Orange
Romain Lettuce
Spinach

Name	Cost	List Pri...
Romain L	1.25	1.99

Using a Non-Drag-Enabled Component in a Drag-and-Drop Operation

So far, you have been taking advantage of some of the enhanced functionality in list-based components with regard to drag and drop. Now it's time to learn how to start drag and drop on non-enhanced components. In this particular task, the use case is very simple: You want to drag a Label control to a DataGrid. Because the Label does not have enhanced drag-and-drop functionality, there is more of a burden on you as the developer to implement it.

Understanding what the list-based components did for you is a good place to start when having to write all the implementation yourself. Here is a list of mechanisms, hidden from you when using the list-based components, that you'll need to use when implementing drag and drop without the help of the enhanced components:


- Assign the data to the `DragSource` object.
- Check to see whether the formats allow dropping onto the drop target.
- Use the data in the drop target (although in the second exercise you did some of this manually).
- Permit the component to be dragged.
- Accept the drop.

Although you have been using the `DragSource` class up to now in this lesson, you'll need to dig deeper into the class when implementing all the functionality yourself. In this exercise, you'll use the following methods of the `DragSource` class:

DragSource Class Methods	
Method	Description
<code>addData(data:Object, format:String):void</code>	Adds data to the associated format in the <code>DragSource</code> object; the * denotes that the data can be of any data type.
<code>hasFormat(format:String):Boolean</code>	Returns <code>true</code> if the <code>DataSource</code> object contains a matching format of the drop target; otherwise, it returns <code>false</code> .
<code>dataForFormat(format:String):Object</code>	Retrieves the data for the specified format added by the <code>addData()</code> method. If there are multiple items for the format it returns an Array of objects containing the data in the requested format. If using Flex's List classes, even a single item is returned in a one-item Array.

These methods allow you to implement the first three hidden mechanisms. To implement the last two, you need to use methods of the `DragManager` class:

DragManager Class Methods	
Method	Description
<code>doDrag(initiator:IUIComponent ➔ dragSource:DragSource, ➔ mouseEvent:MouseEvent):void</code>	Enables the initiator component to be initially dragged; often used in an event handler for <code>mouseDown</code> or <code>mouseMove</code> .
<code>acceptDragDrop(target:Component):void</code>	Calls this method in your <code>dragEnter</code> handler; often used in an <code>if</code> statement where the condition uses the <code>hasFormat()</code> method.

 **TIP:** The `doDrag()` method has a number of optional parameters to control the look of the drag proxy. You can find these parameters in the Class documentation for `DragManager` in the Adobe Flex 4.5 Language Reference.

Now you're ready to start writing code for this exercise.

- 1 Examine the code in the `Task3_Label_to_DataGrid.mxml` file, and then run it.

It is essentially the same as the end of Task 2, however, you'll see you have a Label with the text "Drag me" in it. At this point, the drop handling functionality you wrote in the previous lesson is still in place, but there is no code to handle the drag or drop for the Label.

- 2 Begin adding a `mouseDown` event listener to the Label, and select Generate MouseDown handler to have Flash Builder create the event handler for you. The generated code calls the newly created event handler and passes the event object as a parameter.

- 3 In the `<fx:Script>` block, find the newly created function named `initiatorLabel_mouseDownHandler()`. As the first line of the function, create a local variable named `dragSource` and assign it to a new instance of the `DragSource` class.

This creates the `DragSource` object that will contain the data and information about your drag operation.

- 4 Next, create a new local variable named `product` of type `Object` and assign it to a new instance of `Object`.

```
var product:Object = new Object();
```

- 5 Assign several properties of the product object to initial values, including name to **My Product**, cost to **1.99**, and listPrice to **2.99**:

```
protected function initiatorLabel_mouseDownHandler(event:MouseEvent):void {  
    var dragSource:DragSource = new DragSource();  
  
    var product:Object = new Object();  
    product.name = "My Product";  
    product.cost = 1.99;  
    product.listPrice = 2.99;  
}
```

- 6 Next, in the function, use the `addData()` method of the `dragSource` object to add the product to the `DragSource` instance. Associate it with a format name `singleProduct`.

```
dragSource.addData( product, "singleProduct" );
```

An important point here is that you can store data associated with multiple formats, which means you can use multiple `addData()` methods on the same `DragSource` object. You might want to do this if you have multiple drop targets and want to drop different data in each drop target. For example, maybe you have the same data as text or as a PDF. The different drop targets would use different arguments in the `dataForFormat()` method to get the appropriate data.

In this case, you are associating this product with a format named `singleProduct`.

- 7** As the last line of code in the function, permit the Label to be dragged by calling the static `doDrag()` method of the `DragManager` class. You pass it the three arguments `initiator`, `dragSource`, and `event`. The initiator will be the `event.target` (the label) cast as an `IUIComponent`, and the other two arguments should be obvious. Check to make sure your completed function appears as shown here:

```
protected function initiatorLabel_mouseDownHandler(event:MouseEvent):void {
    var dragSource:DragSource = new DragSource();

    var product:Object = new Object();
    product.name = "My Product";
    product.cost = 1.99;
    product.listPrice = 2.99;

    dragSource.addData( product, "singleProduct" );
    DragManager.doDrag( event.target as IUIComponent, dragSource, event );
}}
```

Remember that a static method is one you can invoke directly from the class without first instantiating it.

- 8** Run the application and drag the Label. Although you can drag it, there are no components registered to accept data of type `singleProduct`, so you cannot drop it anywhere.

You can now code the `DataGrid` to accept the drop of the Label and to display the data passed in the `DragSource` in the `DataGrid`.

- 9** Find the `targetGrid_dragEnterHandler()`. Right now this method only accepts a drag if your data has a format named `itemsByIndex`. Your data has a format named `singleProduct`. Change this function to accept `itemsByIndex` or `singleProduct` as show in the following code:

```
protected function targetGrid_dragEnterHandler(event:DragEvent):void {
    var dragSource:DragSource = event.dragSource;

    if ( dragSource.hasFormat( "itemsByIndex" ) ||
        dragSource.hasFormat( "singleProduct" ) ) {
        DragManager.acceptDragDrop( event.target as IUIComponent );
    }
}
```

This function will now allow the `DataGrid` to accept data in either format. If you run the application now and drag the label onto the `DataGrid`, the drag will be accepted, but the code will crash as the `DataGrid` is accepting data in the form of a `Vector` and your `singleProduct` format only contains one product. You will fix that next.

- 10** Find the `targetGrid_dragDropHandler()` method. Right now this method assumes that it will always receive a `Vector` in the `DragSource` object. That is no longer a valid assumption. After the two variable declarations, wrap the next two lines of code in an `if` statement that checks if the `dragSource` has a format named `itemsByIndex`.

```
protected function targetGrid_dragDropHandler(event:DragEvent):void {
    var items:Vector.<Object>;
    var dragSource:DragSource = event.dragSource;

    if ( dragSource.hasFormat( "itemsByIndex" ) ) {
        items = dragSource.dataForFormat( "itemsByIndex" ) as Vector.<Object>;
        targetListDP.addItem( items[ 0 ] );
    }
}
```

Now this code checks to make sure there is a format named `itemsByIndex` before it attempts to retrieve that data.

- 11** Next, add an `else-if` clause to the same block that checks to see if the data has a format named `singleProduct`:

```
protected function targetGrid_dragDropHandler(event:DragEvent):void {
    var items:Vector.<Object>;
    var dragSource:DragSource = event.dragSource;

    if ( dragSource.hasFormat( "itemsByIndex" ) ) {
        items = dragSource.dataForFormat( "itemsByIndex" ) as Vector.<Object>;
        targetListDP.addItem( items[ 0 ] );
    } else if ( dragSource.hasFormat( "singleProduct" ) ) {
    }
}
```

- 12** Finally, inside the `else` block, call the `targetListDP.addItem()` method and pass it the result of the `dragSource.dataFormat()` method with `singleProduct` as the argument.

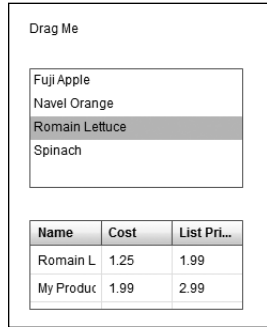
```
protected function targetGrid_dragDropHandler(event:DragEvent):void {
    var items:Vector.<Object>;
    var dragSource:DragSource = event.dragSource;

    if ( dragSource.hasFormat( "itemsByIndex" ) ) {
        items = dragSource.dataForFormat( "itemsByIndex" ) as Vector.<Object>;
        targetListDP.addItem( items[ 0 ] );
    } else if ( dragSource.hasFormat( "singleProduct" ) ) {
        targetListDP.addItem( dragSource.dataForFormat( "singleProduct" ) );
    }
}
```

At this point, your `DataGrid` will accept a drag operation with data formatted as either `itemsByIndex` or as a `singleProduct` and will handle those two situations differently.

13 Save and run the application.

You should now be able to drag the Label. When it moves over the DataGrid, the red X disappears, and you can drop the drag proxy. You can also continue to drag items from the List to the DataGrid.



Now that you have a solid background in drag and drop, you will implement drag-and-drop functionality in the e-commerce application of FlexGrocer.

Dragging a Grocery Item to the Shopping Cart

The culmination of your work in this lesson is to implement dragging a grocery item into the shopping cart. The exercises you have performed so far in this lesson have prepared you well for this final exercise.

In these steps, you will enable the user to click the grocery item, drag it to the small shopping cart, and then drop it in. The grocery item is displayed in a VGroup container, and the shopping cart is a List. Because the VGroup is not a drag-and-drop-enhanced component, you will have to pattern your code here after what you just wrote in the section “Using a Non-Drag-Enabled Component in a Drag-and-Drop Operation.”

1 Open the FlexGrocer project you used in the previous lesson.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson13/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

2 Open ProductItem.mxml from the components package.

This is the component in which the grocery data is displayed; so this is where you'll have to permit the data to be dragged.

► **TIP:** At first, you'll drag all the data to the shopping cart and then write the code so that just the image of the item acts as the drag proxy.

- 3 Locate the `<s:VGroup>` just below the `<fx:Declarations>` tag set. In that container, locate the `<s:Image>` tag that displays the product. Add an `id` property to the `Image` tag and assign it the value `img`.

You will be referencing this image several times and need to give it an instance name.

- 4 Add a `mouseDown` event in the `Image` tag and select `Generate DragDrop` handler to have Flash Builder create the event handler for you. Note that the event object was automatically passed.

```
mouseDown="img_mouseDownHandler(event)"
```

By placing the `mouseDown` event on the `Image`, you will enable the user to start the drag process by clicking the image.

- 5 Locate the newly created event handler in the `<fx:Script>` block and remove the `TODO` comment.

```
protected function img_mouseDownHandler(event:MouseEvent):void {  
}
```

This function has two main purposes: to permit the component to be dragged and to store appropriate data into the object being dragged.

- 6 As the first line of code in the event handler, create a new variable local to the function named `dragSource` and assign it a new `DragSource` object. Next, use the `addData()` method to associate the newly created object with the product and a format named `singleProduct`.

```
var dragSource:DragSource=new DragSource();  
dragSource.addData( product,"singleProduct");
```

If you used code-completion, `DragSource` was imported for you. Otherwise, import `mx.core.DragSource` manually. Remember that the `addData()` method's two parameters assign the data and the format to the `DragSource` object. In this case the data is the product being displayed in the `VGroup`, and the format is a description of the format of the data in the drag source.

- 7 As the last line of code in the function, permit the `Image` to be dragged by calling the static `doDrag()` method of the `DragManager` class. Recall that you must pass the method three parameters, the initiator, the `DragSource` object, and the event. In this case, the initiator is `event.currentTarget as IUIComponent`. The `DragSource` is the `dragSource` object you created. Lastly, the event is the event parameter passed to the event handler.

```
DragManager.doDrag(event.currentTarget as IUIComponent, dragSource, event);
```

If you used code-completion, `IUIComponent` and `DragManager` were imported for you. If not, import `mx.core.IUIComponent` and `mx.managers.DragManager` now. You had to cast the initiator as `IUIComponent` because `event.currentTarget` is typed as an `Object` by default, which is not a visual component and hence cannot be dragged.

- 8** Run the `FlexGrocer.mxml` application. You should be able to drag the grocery item data.

You'll see the drag proxy is the outline of the `Image`, or a rectangular box. Later in this task, you will change the drag proxy to the image of the grocery item.

At this point there is no drop target, so you cannot drop the data anywhere.

- 9** Open the `ShoppingView.mxml` from the `views` package.

This file contains the `List` that is the shopping cart to which grocery items are dragged.

- 10** Locate the `List` with the `id` of `cartList` and add a `dragEnter` event to the `List`. Select `Generate DragDrop` handler to have Flash Builder create the event handler for you.

```
dragEnter="cartList_dragEnterHandler(event)"
```

- 11** Locate the newly generated event handler and remove the `TODO` comment. Add a new local variable named `dragSource` of type `DragSource` and assign it to the `event.dragSource`.

```
protected function cartList_dragEnterHandler(event:DragEvent):void {
    var dragSource:DragSource = event.dragSource;

}
```

- 12** Next, check if the `dragSource` has a format of `singleProduct` using the `dragSource`'s `hasFormat()` method. If it does, then use the `DragManager`'s `acceptDragDrop()` method, passing it a reference to the event's target cast as an `IUIComponent`. Your method should look like this:

```
protected function cartList_dragEnterHandler(event:DragEvent):void {
    var dragSource:DragSource = event.dragSource;

    if ( dragSource.hasFormat( "singleProduct" ) ) {
        DragManager.acceptDragDrop( event.target as IUIComponent );
    }
}
```

This function has only one purpose: to check whether formats enable the drag initiator to be dropped. The `if` statement determines whether there are matching formats; then the `acceptDragDrop()` method allows the actual dropping to take place.

- 13** Run the `FlexGrocer.mxml` application. You should be able to drag the grocery item data; when you drag the pointer over the shopping cart `List`, you should see the red `X` disappear, and you can drop the drag proxy.

At this point, nothing happens when you drop the drag proxy.

- 14** In `ShoppingView.xml`, locate the List with the id of `cartList` and add a `dragDrop` event to the List. Select Generate `DragDrop` handler to have Flash Builder create the event handler for you.

```
dragDrop="cartList_dragDropHandler(event)"
```

- 15** Locate the newly generated event handler and remove the `TODO` comment.

```
protected function cartList_dragDropHandler(event:DragEvent):void {
}
```

- 16** As the first line of code in the function, create a variable local to the function named `product`, typed as a `Product`. Assign `product` the result of calling the `dataForFormat()` method of the `event.dragSource` object, passing `singleProduct` as the argument. You will need to cast the result as a `Product`.

```
var product:Product = event.dragSource.dataForFormat(
    ↪ "singleProduct" ) as Product;
```

This `Product` object is needed to create a `ShoppingCartItem` in the next step of the task. The `dataForFormat()` method retrieves data based on the `singleProduct` format. In this case, the data stored in the `DragSource` object was the product data added in step 6 of this task using the `addData()` method.

- TIP:** You can open the file `cart/ShoppingCartItem.as` and review that the constructor's parameters are a `Product` object and an optional quantity.

- 17** Next in the function, create a variable local to the function named `shoppingCartItem`, typed as a `ShoppingCartItem`. Assign that variable equal to a new `ShoppingCartItem`. The arguments of the `ShoppingCartItem` constructor should be the `Product` object created in the last step and the number 1.

```
var shoppingCartItem:ShoppingCartItem = new ShoppingCartItem(product,1);
```

Here is a quick review of how the `Product` object got in the `DragSource`:

- In steps 4, 5 and 6 of this exercise, you passed a `Product` object to the `img_mouseDownHandler()` function.
 - The function placed the `Product` object into the `DragSource` object using the `addData()` method and associated it with the `singleProduct` format.
 - In the event handler just created, you retrieved that same `Product` object and will now place it in the shopping cart.
- 18** As the last line of code in the function, invoke the `addItem()` method of the `shoppingCart` object and pass the `shoppingCartItem` variable as a parameter.

Check to be sure your function appears as shown here:

```
protected function cartList_dragDropHandler(event:DragEvent,
    ➤ format:String):void{
    var product:Product = event.dragSource.dataForFormat( format ) as Product;
    var shoppingCartItem:ShoppingCartItem =
    ➤ new ShoppingCartItem(product,1);
    shoppingCart.addItem( shoppingCartItem );
}
```

The method invocation actually places the `ShoppingCartItem` object in the shopping cart.

- 19** Run the application. You can now drag grocery items into the shopping cart.

You'll see that the drag-and-drop operation is working, but the drag proxy is the default proxy for the `Image`. In the next step you add code so the drag proxy becomes the image of the grocery item.

- 20** Return to `ProductItem.mxml`. At the bottom of the `<fx:Script>` block, locate the `img_mouseDownHandler()` function. At the top of the function create a variable local to the function named `proxy` of type `Image`. Assign the newly created variable a new `Image` object. As the next line of code in the function, assign `proxy.source` the value `img.bitmapData`.

```
var proxy:Image = new Image();
proxy.source = img.bitmapData;
```

Here you are creating a new image to act as the proxy. This code may cause you to have some questions.

Your first question might be: Why not just use the existing `Image` object as the proxy? This is because by default the drag-and-drop operation removes the drag proxy from its source. You could have simply used the existing `Image` as the drag proxy, but after dragging and dropping, the image would no longer be shown with the other grocery item data.

Second, why are you assigning the source property of the proxy to the `bitmapData` of the images? The `bitmapData` represents the raw bitmap data needed to represent this image on the screen. Rather than downloading a copy or retrieving it from the browser's cache, you are simply indicating that the new image should use the same chunk of memory to display this image.

- 21** In the `DragManager.doDrag()` method invocation, add a fourth parameter of proxy.

```
DragManager.doDrag( event.currentTarget as IUIComponent,
    ➤ dragSource, event, proxy);
```

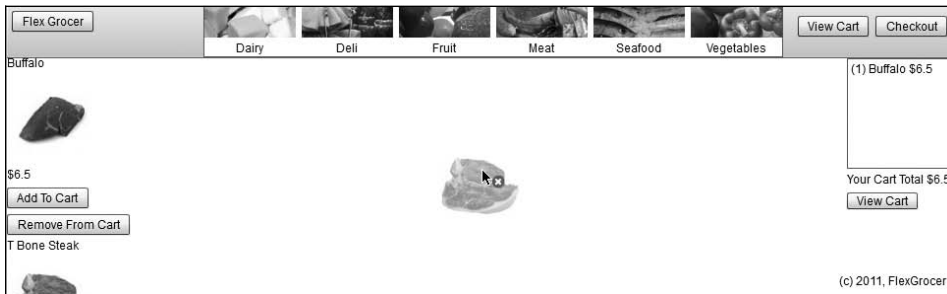
This fourth parameter represents the `dragImage`. Instead of the outline of the `Image` of the grocery item data being the drag proxy, you have now specified that the image of the item should be displayed when dragging.

- 22** Check to be sure that your `img_mouseDownHandler()` function appears as follows, and then save and run the application. You should be able to drag the image of the grocery item and drop it in the cart.

```
protected function img_mouseDownHandler(event:MouseEvent):void {
    var proxy:Image = new Image();
    proxy.source = img.bitmapData;

    var dragSource:DragSource = new DragSource();
    dragSource.addData( product, "singleProduct" );

    DragManager.doDrag( event.currentTarget as IUIComponent,
        ➤ dragSource, event, proxy );
}
```



What You Have Learned

In this lesson, you have:

- Learned the basics about the Drag and Drop Manager (pages 312-313)
- Implemented drag-and-drop operations using the default drop process (pages 313–315)
- Implemented drag-and-drop using a customized the drop process to use the data stored in the DragSource object (pages 315–321)
- Implemented drag-and-drop operations between non-drag-enabled components and used a custom `dragImage` (pages 321–326)
- Implemented drag-and-drop operations in the shopping cart (pages 326–331)

LESSON 14

What You Will Learn

In this lesson, you will:

- Use states as the basis for implementing navigation
- Learn about two-way binding
- Work with the Form class

Approximate Time



This lesson takes approximately 1 hour and 30 minutes to complete.

LESSON 14

Implementing the Checkout Process

Imperative to any application is a navigation system. Users should be able to move easily around in an application and locate the functionality they need. Some navigation will be completely at the user's discretion, such as clicking a button to move to the home page or the checkout process. Other navigation can be tightly controlled by the developer—for example, a checkout process in which users cannot proceed to the next screen until certain conditions are met on an existing screen.

The screenshot shows a web application interface for FlexGrocer. At the top, there is a navigation bar with a 'Flex Grocer' logo and a menu of categories: Dairy, Deli, Fruit, Meat, Seafood, and Vegetables. To the right of the menu are 'View Cart' and 'Checkout' buttons. Below the navigation bar, the page is titled 'Checkout Page 3 of 3'. On the left side, there is a 'Review and Checkout' section with the following details: Name: ima flexuser, Address: 715 any lane nyc, ny 10001, and Card Type: American Express. Below this information are two buttons: 'Complete Order' and 'Edit Information'. On the right side, there is a table listing the items in the cart. The table has columns for Product, Quantity, and Amount. Each row includes a small image of the product, a 'Remove' button, and a 'Total 10.07' label at the bottom of the table. The copyright notice '(c) 2011, FlexGrocer' is located at the bottom right of the page.

Product	Quantity	Amount	
 Bananas	1	\$1.98	<input type="button" value="Remove"/>
 Buffalo	1	\$6.5	<input type="button" value="Remove"/>
 Milk	1	\$1.59	<input type="button" value="Remove"/>

Total 10.07

(c) 2011, FlexGrocer

The checkout process will be controlled by states.

Introducing Navigation with States

In Lesson 3, “Laying Out the Interface,” you learned how you can use states to change the appearance of a component at runtime. In this lesson, you’ll take the same states idea, but instead of just changing the appearance, you’ll change which of several components are shown to the user, so that they can see only one child at a time. To implement this, you’ll use a container with a basic layout (so the children can all be positioned at the 0, 0 point), and use the `includeIn` property, so only one of the children is available in each state.

Introducing Two-Way Bindings

In Lesson 8, “Using Data Binding and Collections,” you learned about data binding, which updates the view components when data changes. Flex 4 introduces a second type of data binding called two-way binding. Two-way binding is most effective when combined with user input forms.

In this type of data binding, view components are updated when data changes; however, the data is also updated when the component changes. You can think of regular data binding as moving in one direction: When the data changes, the view changes. You can think of two-way binding as bidirectional: If either changes, the other updates.

In practice, two-way binding is extremely easy to use. There is simply a syntax difference when declaring a control as bound to data.

To bind a `TextInput` to a piece of data using traditional data binding, your code would look like this:

```
<s:TextInput text="{someData}"/>
```

To use two-way binding, you simply prepend the expression with the `@` symbol:

```
<s:TextInput text="@{someData}"/>
```

Two-way binding does have one limitation at this time: The types of both the source and destination must be the same. With traditional data binding, you can bind a variable declared as a `Number` to the text input of a `Label`, even though that `Label` is expecting a `String` instance. Traditional data binding will attempt to convert the `Number` to a `String` on your behalf. Two-way data binding cannot work unless both the source and destination are the same.

Creating the OrderInfo valueObject

Throughout this chapter, you'll be building the checkout process for the FlexGrocer application. In this process, users will be presented with a series of components to allow them to enter various types of information (shipping info, billing info) and a confirmation screen. The information users enter will be stored in a value object. Using two-way data binding, the value object can be populated automatically as the user fills out the form. The first step in this process is to create the OrderInfo object.

- 1 Open the FlexGrocer.mxml file that you created in the previous lesson.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson14/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 In the declarations section, create an instance of the ShoppingCart class, and give it an id of shoppingCart.

```
<fx:Declarations>
<!-- Place non-visual elements (e.g., services, value objects) here -->
  <services:CategoryService id="categoryService"/>
  <services:ProductService id="productService"/>
  <cart:ShoppingCart id="shoppingCart"/>
</fx:Declarations>
```

You'll need access to the same shopping cart from both the shopping and checkout experience, so you'll create it here in the main application, and pass it into the places where it is used.

- 3 Find the place where the ShoppingView is instantiated; populate its shoppingCart property with a binding to the shoppingCart instance you created in the previous step.

```
<views:ShoppingView id="bodyGroup"
  width="100%" height="100%"
  groceryInventory="{productService.products}"
  shoppingCart="{shoppingCart}" />
```

- 4 Open ShoppingView.mxml. Find the property declaration for the public shoppingCart property. Remove the equal sign and everything after it, so the property is declared, but not instantiated:

```
public var shoppingCart:ShoppingCart;
```

Since the cart is now passed in from the main application, you no longer want to create it here.

5 Right-click on the `valueObjects` package, and choose **New ActionScript Class**. Leave the package as **valueObjects**, and name the class **OrderInfo**. Leave the other elements with the default values, and click **Finish**.

6 Decorate the class with the `Bindable` metadata.

```
package valueObjects {
    [Bindable]
    public class OrderInfo {
        public function OrderInfo() {
        }
    }
}
```

7 Before the constructor, define public properties for `billingName`, `billingAddress`, `billingCity`, `billingState`, `billingZip`, `cardType`, `cardNumber`, `cardExpirationMonth`, and `cardExpirationYear`. Define all nine of these properties to use a `String` as a data type.

```
package valueObjects {
    [Bindable]
    public class OrderInfo {
        public var billingName:String;
        public var billingAddress:String;
        public var billingCity:String;
        public var billingState:String;
        public var billingZip:String;
        public var cardType:String;
        public var cardNumber:String;
        public var cardExpirationMonth:String;
        public var cardExpirationYear:String;

        public function OrderInfo() {
        }
    }
}
```

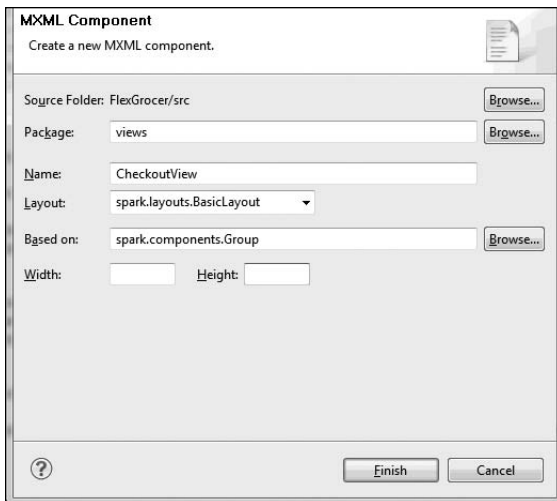
8 Save and close the `OrderInfo`.

This completes the `OrderInfo` class. In the next exercise, you'll create the `CheckoutView` component, which will use an instance of this class.

Creating CheckoutView

Next, you'll create the CheckoutView component, which contains the views a user will experience in the checkout process.

- 1 Right-click the views package and create a new MXML Component. Leave the Package set as **views**, give the component a Name of **CheckoutView**, set the Layout to be **spark.layouts.BasicLayout**, leave the Based on value as **spark.components.Group**, and remove the values from the Width and Height fields.



- 2 After the end of the layout block, create a states block to hold the states. You'll use the `<s:states>` tag (with a lower case s in states), as you are addressing the states property of this new component.

```
<s:layout>
  <s:BasicLayout/>
</s:layout>
<s:states>

</s:states>
```

- 3 Add three `<s:State>` (capital S) instances to define the states for the component. The first should have a name of `customerInfo`, the second `billingInfo`, and the third `review`.

```
<s:states>
  <s:State name="customerInfo"/>
  <s:State name="billingInfo"/>
  <s:State name="review"/>
</s:states>
```


Here you are defining the three states of this component, which will show the individual views of the Checkout process.

- 4 In the Declarations block, create an instance of the valueObject of OrderInfo. Give the instance an id of orderInfo.

```
<fx:Declarations>
  <!-- Place non-visual elements (e.g., services, value objects) here -->
  <valueObjects:OrderInfo id="orderInfo"/>
</fx:Declarations>
```

You now have an instance of the OrderInfo class that can be populated by the views to which it is passed. Next you'll add a few methods to ease the process of navigating between the views.

- 5 Create a Script block, and a private variable named currentView, data typed as an int, with a default value of 0.

```
</s:states>
<fx:Script>
  <![CDATA[
    private var currentView:int=0;
  ]]>
</fx:Script>
<fx:Declarations>
```

This will be used to keep track of which views users are on, so when they click a next button, you know what view to show them next.

- 6 Create a private method named setViewByIndex, which accepts an integer as a parameter and returns void. In the method, create a switch statement that will set the currentState to customerInfo if the value 1 is passed, billingInfo if a 2 is passed, and review if 3 is passed in.

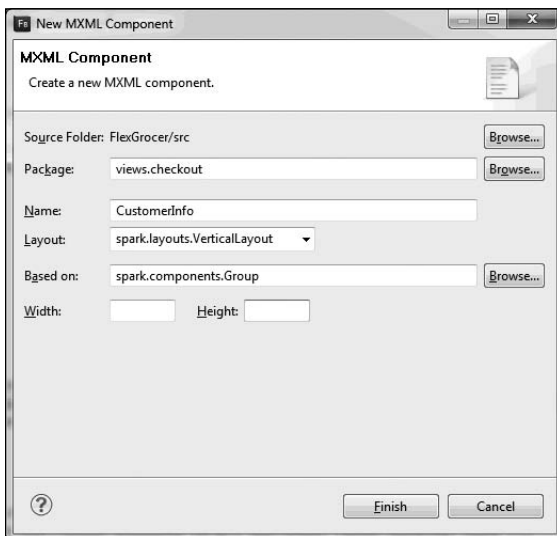
```
private function setViewByIndex(index:int):void{
  switch(index){
    case 0:
      currentState="customerInfo";
      break;
    case 1:
      currentState="billingInfo";
      break;
    case 2:
      currentState="review";
      break;
  }
}
```

- 7 Create a private method called `handleProceed` that accepts an event as an argument, and returns `void`. In this method, increment the value of `currentView`, and then pass `currentView` to the `setViewByIndex` method.

```
private function handleProceed( event:Event ):void {  
    currentView++;  
    setViewByIndex(currentView);  
}
```

Now, whenever a `proceed` event is heard, the `currentState` will be changed to the next state in the order. Next, you'll create the `CustomerInfo` view, and set it to be shown in the proper state.

- 8 Right-click the `views` package and choose `New MXML Component`. Set the Package to be `views.checkout`, set the Name to be `CustomerInfo`, set the Layout to `spark.layouts.VerticalLayout`, and leave the Based on as `spark.components.Group`. Remove the Width and Height values. Click `Finish`.



- 9 Add a `Label` after the closing `Declaration` tag. Set the text of the `Label` to be `Checkout`

- 10** Add a Script block after the declarations but before the Label. Declare a public bindable property named `orderInfo` data typed as `valueObject` of the `OrderInfo`.

```
<fx:Declarations>
  <!-- Place non-visual elements (e.g., services, value objects) here -->
</fx:Declarations>
<fx:Script>
  <![CDATA[
    import valueObjects.OrderInfo;
    [Bindable]
    public var orderInfo:OrderInfo;
  ]]>
</fx:Script>
<s:Label text="Checkout Page 1 of 3"/>
```

- 11** Open `Checkout.mxml` from the default package. Copy the entire form from the component. Switch back to `CustomerInfo.mxml`, and paste the form after the `Label` tag. Remove the `x` and `y` values from the `<s:Form>` tag.

```
<s:Label text="Checkout Page 1 of 3"/>
<s:Form >
  <s:FormHeading label="Customer Information"/>
  <s:FormItem label="Customer Name">
    <s:TextInput/>
  </s:FormItem>
  <s:FormItem label="Address">
    <s:TextInput/>
  </s:FormItem>
  <s:FormItem label="City">
    <s:TextInput/>
  </s:FormItem>
  <s:FormItem label="State">
    <s:TextInput/>
  </s:FormItem>
  <s:FormItem label="Zip">
    <s:TextInput/>
  </s:FormItem>
  <s:FormItem>
    <s:Button label="Continue"/>
  </s:FormItem>
</s:Form>
```

- 12** Find the `TextInput`, which is a child of the first form item (which has a label `Customer Name`). Set the `text` property of this `TextInput` to be a two-way binding to the `billingName` property of the `orderInfo` object.

```
<s:TextInput text="@{orderInfo.billingName}"/>
```

As discussed earlier in the lesson, this syntax will create a two-way binding, so the control will initially be populated with the value of the `billingName` property of the `orderInfo` object, and if the user makes a change, the change will be stored automatically back in the `orderInfo` object.

- 13** Add similar two-way bindings for the text inputs in the Address, City, State, and Zip FormItems. The values you'll bind to are `billingAddress`, `billingCity`, `billingState`, and `billingZip`.

```
<s:Form >
  <s:FormHeading label="Customer Information"/>
  <s:FormItem label="Customer Name">
    <s:TextInput text="@{orderInfo.billingName}"/>
  </s:FormItem>
  <s:FormItem label="Address">
    <s:TextInput text="@{orderInfo.billingAddress}"/>
  </s:FormItem>
  <s:FormItem label="City">
    <s:TextInput text="@{orderInfo.billingCity}"/>
  </s:FormItem>
  <s:FormItem label="State">
    <s:TextInput text="@{orderInfo.billingState}"/>
  </s:FormItem>
  <s:FormItem label="Zip">
    <s:TextInput text="@{orderInfo.billingZip}"/>
  </s:FormItem>
  <s:FormItem>
    <s:Button label="Continue"/>
  </s:FormItem>
</s:Form>
```

Now your whole form is set to be bound to the billing properties of `orderInfo`.

- 14** Add a `click` handler on the Continue button, which calls the `handleProceed` method and passes the event as an argument. With your cursor after the closing parentheses, which are after the word `event`, press `Ctrl-I` and choose `Generate event handler`.



- 15** In the newly created event handler, dispatch an event with the type `proceed`.

```
protected function handleProceed(event:MouseEvent):void{
    dispatchEvent( new Event( "proceed" ) );
}
```

- 16** As a final step for this component, add metadata indicating that this class is capable of broadcasting an event with the name `proceed` and a type of `flash.events.Event`.

```
<fx:Metadata>
    [Event(name="proceed", type="flash.events.Event")]
</fx:Metadata>
```

- 17** Save your file.

This completes the `CustomerInfo` component. The complete source for this component should appear like the following code.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <fx:Metadata>
        [Event(name="proceed", type="flash.events.Event")]
    </fx:Metadata>
    <fx:Script>
        <![CDATA[
            import valueObjects.OrderInfo;
            [Bindable]
            public var orderInfo:OrderInfo;

            protected function handleProceed(event:MouseEvent):void{
                dispatchEvent( new Event( "proceed" ) );
            }

        ]]>
    </fx:Script>
    <s:Label text="Checkout Page 1 of 3"/>
    <s:Form >
        <s:FormHeading label="Customer Information"/>
        <s:FormItem label="Customer Name">
            <s:TextInput text="@{orderInfo.billingName}"/>
        </s:FormItem>
        <s:FormItem label="Address">
            <s:TextInput text="@{orderInfo.billingAddress}"/>
        </s:FormItem>
        <s:FormItem label="City">
            <s:TextInput text="@{orderInfo.billingCity}"/>
        </s:FormItem>
        <s:FormItem label="State">
            <s:TextInput text="@{orderInfo.billingState}"/>
        </s:FormItem>
    </s:Form >
</s:Group>
```

```

</s:FormItem>
<s:FormItem label="Zip">
  <s:TextInput text="@{orderInfo.billingZip}"/>
</s:FormItem>
<s:FormItem>
  <s:Button label="Continue" click="handleProceed(event)"/>
</s:FormItem>
</s:Form>
</s:Group>

```

Next, you'll instantiate this component in the CheckoutView component.

- 18** Open CheckoutView.mxml. After the closing Declarations tag, add an instance of the CustomerInfo component. Assign a width and height of 100%, bind the orderInfo object to the orderInfo property of the component, listen for the proceed event, and call the handleProceed method you wrote in a previous exercise. Lastly, set this component to be shown only in the customerInfo state, by using the includeIn attribute. The final code for CheckoutView should read like this:

```

<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark" width="400" height="300" xmlns:
s:valueObjects="valueObjects.*" xmlns:checkout="views.checkout.*">
  <s:layout>
    <s:BasicLayout/>
  </s:layout>
  <s:states>
    <s:State name="customerInfo"/>
    <s:State name="billingInfo"/>
    <s:State name="review"/>
  </s:states>
  <fx:Script>
    <![CDATA[
      private var currentView:int=0;
      private function setViewByIndex(index:int):void{
        switch(index){
          case 0:
            currentState="customerInfo";
            break;
          case 1:
            currentState="billingInfo";
            break;
          case 2:
            currentState="review";
            break;
        }
      }
      private function handleProceed( event:Event ):void {

```

```

        currentView++;
        setViewByIndex(currentView);
    }
    ]]>
</fx:Script>
<fx:Declarations>
    <!-- Place non-visual elements (e.g., services, value objects) here -->
    <valueObjects:OrderInfo id="orderInfo"/>
</fx:Declarations>
<checkout:CustomerInfo width="100%" height="100%"
    orderInfo="{orderInfo}"
    proceed="handleProceed( event )"
    includeIn="customerInfo"/>
</s:Group>

```

- 19** Save your file. CheckoutView is now equipped to work with your new component. The next step is to instantiate CheckoutView in the main application.

- 20** Open FlexGrocer.mxml. After the closing declarations tag, add two states: one called shopping and the other called checkout.

```

<s:states>
    <s:State name="shopping"/>
    <s:State name="checkout"/>
</s:states>

```

- 21** Just after the bodyGroup instance of the ShoppingView component, create an instance of your CheckoutView component. Set the width and height to 100%. Set the includeIn attribute to only show this component in the checkout state. Set the includeIn for the ShoppingView to be shopping.

```

<views:ShoppingView id="bodyGroup"
    width="100%" height="100%"
    groceryInventory="{productService.products}"
    shoppingCart="{shoppingCart}"
    includeIn="shopping"/>
<views:CheckoutView width="100%" height="100%"
    includeIn="checkout"/>

```

- 22** Create a new method called startCheckout that accepts a MouseEvent as a parameter and returns void. The body of the function should set the currentState to checkout. Call this function from the click handler of btnCheckout.

```

protected function startCheckout(event:MouseEvent):void{
    this.currentState="checkout";
}
...
<s:Button id="btnCheckout" y="10" right="10" label="Checkout"
click="startCheckout(event)"/>

```

- 23** Save and run the application. Now, if you click the Checkout button, the application will change states to show the CustomerInfo component.



Now your shopping application allows the user to add products to the shopping cart and to move to the checkout process.

Creating CreditCardInfo

The next step is to add a second page to the checkout process, which will allow the user to enter their billing info.

- 1 Right-click the checkout package, and choose new MXML Component. Leave the Package as **views.checkout**, assign a name of **CreditCardInfo**, set the Layout to **spark.layouts.VerticalLayout**, and leave the Based on as **spark.components.Group**. Remove the Width and Height values. Click Finish.
- 2 Copy the Metadata and Script block from the CustomerInfo component, and paste them in the CreditCardInfo component, after the Declarations block.

```
<fx:Metadata>
    [Event(name="proceed", type="flash.events.Event")]
</fx:Metadata>
<fx:Script>
    <![CDATA[
        import valueObjects.OrderInfo;
        [Bindable]
        public var orderInfo:OrderInfo;

        protected function handleProceed(event:MouseEvent):void{
            dispatchEvent( new Event( "proceed" ) );
        }

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


Both components will use the `orderInfo` object, and both will broadcast a `proceed` event to indicate the user has finished with this page.

- 3** After the closing `Script` tag, add a label with its `text` property set to “Checkout Page 2 of 3”.

```
<s:Label text="Checkout Page 2 of 3"/>
```

- 4** Add a `Form` tag after the label. Add a `FormHeading` that reads “Billing Information”. Add four form items. Their labels should be `Credit Card Type`, `Card Number`, and `Expiration`. The final `FormItem` should have no label.

```
<s:Label text="Checkout Page 2 of 3"/>
<s:Form>
  <s:FormHeading label="Billing Information"/>
  <s:FormItem label="Credit Card Type">
  </s:FormItem>
  <s:FormItem label="Card Number">
  </s:FormItem>
  <s:FormItem label="Expiration">
  </s:FormItem>
  <s:FormItem>
  </s:FormItem>
</s:Form>
```

- 5** In the first `FormItem`, create a `DropDownList`. Use two-way binding to bind the `selectedItem` property to the `orderInfo`’s `cardType` property. Set the `requireSelection` property to `true`. Add a child tag to the `DropDownList` to address the `dataProvider` property. Inside the `dataProvider`, create an `ArrayList` that contains these five strings: `American Express`, `Diners Club`, `Discover`, `Mastercard`, and `Visa`.

```
<s:FormItem label="Credit Card Type">
  <s:DropDownList selectedItem="@{orderInfo.cardType}"
  requireSelection="true">
    <s:dataProvider>
      <s:ArrayList>
        <fx:String>American Express</fx:String>
        <fx:String>Diners Club</fx:String>
        <fx:String>Discover</fx:String>
        <fx:String>MasterCard</fx:String>
        <fx:String>Visa</fx:String>
      </s:ArrayList>
    </s:dataProvider>
  </s:DropDownList>
</s:FormItem>
```

- 6** In the second `FormItem` (the one with the `Card Number` label), create a `TextInput` whose `text` property has a two-way binding to the `cardNumber` property of `orderInfo`:

```
<s:FormItem label="Card Number">
  <s:TextInput text="@{orderInfo.cardNumber}"/>
</s:FormItem>
```

- 7** The third FormItem will have two DropDownLists: one to display months, and one to display years. To allow these to be horizontally next to each other, first add a HorizontalLayout to the FormItem.

```
<s:FormItem label="Expiration">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
</s:FormItem>
```

- 8** Next, add the months DropDownList. Use two-way binding to bind the selectedItem property to the cardExpirationMonth property of orderInfo. Set requireSelection to true. For a dataProvider, create an array list that contains strings for the names of the twelve months.

```
<s:DropDownList selectedItem="@{orderInfo.cardExpirationMonth}"
  requireSelection="true">
  <s:dataProvider>
    <s:ArrayList>
      <fx:String>January</fx:String>
      <fx:String>February</fx:String>
      <fx:String>March</fx:String>
      <fx:String>April</fx:String>
      <fx:String>May</fx:String>
      <fx:String>June</fx:String>
      <fx:String>July</fx:String>
      <fx:String>August</fx:String>
      <fx:String>September</fx:String>
      <fx:String>October</fx:String>
      <fx:String>November</fx:String>
      <fx:String>December</fx:String>
    </s:ArrayList>
  </s:dataProvider>
</s:DropDownList>
```

- 9** Still in the Expiration FormItem, add a second DropDownList. Use two-way binding to bind the selectedItem property to the cardExpirationYear property of orderInfo. Set requireSelection to true. For a dataProvider, create an array list that contains strings with the years 2011 through 2016.

```
<s:DropDownList selectedItem="@{orderInfo.cardExpirationYear}"
  requireSelection="true">
  <s:dataProvider>
    <s:ArrayList>
      <fx:String>2011</fx:String>
      <fx:String>2012</fx:String>
      <fx:String>2013</fx:String>
      <fx:String>2014</fx:String>
    </s:ArrayList>
  </s:dataProvider>
</s:DropDownList>
```

```

        <fx:String>2015</fx:String>
        <fx:String>2016</fx:String>
    </s:ArrayList>
</s:dataProvider>
</s:DropDownList>

```

- 10** In the final `FormItem`, add a button with the label `Proceed`, and a `clickHandler` that calls the `handleProceed` method and passes the event as an argument. The full `CreditCardInfo` component should read like this:

```

<?xml version="1.0" encoding="utf-8"?>
<s:Group xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
    </fx:Declarations>
    <fx:Metadata>
        [Event(name="proceed", type="flash.events.Event")]
    </fx:Metadata>
    <fx:Script>
        <![CDATA[
            import valueObjects.OrderInfo;
            [Bindable]
            public var orderInfo:OrderInfo;
            protected function handleProceed(event:MouseEvent):void{
                dispatchEvent( new Event( "proceed" ) );
            }
        ]]>
    </fx:Script>
    <s:Label text="Checkout Page 2 of 3"/>
    <s:Form>
        <s:FormHeading label="Billing Information"/>
        <s:FormItem label="Credit Card Type">
            <s:DropDownList selectedItem="@{orderInfo.cardType}"
                requireSelection="true">
                <s:dataProvider>
                    <s:ArrayList>
                        <fx:String>American Express</fx:String>
                        <fx:String>Diners Club</fx:String>
                        <fx:String>Discover</fx:String>
                        <fx:String>MasterCard</fx:String>
                        <fx:String>Visa</fx:String>
                    </s:ArrayList>
                </s:dataProvider>
            </s:DropDownList>
        </s:FormItem>
    </s:Form>

```

```
</s:DropDownList>
</s:FormItem>
<s:FormItem label="Card Number">
  <s:TextInput text="@{orderInfo.cardNumber}"/>
</s:FormItem>
<s:FormItem label="Expiration">
  <s:layout>
    <s:HorizontalLayout/>
  </s:layout>
  <s:DropDownList selectedItem=
    "@{orderInfo.cardExpirationMonth}" requireSelection="true">
    <s:dataProvider>
      <s:ArrayList>
        <fx:String>January</fx:String>
        <fx:String>February</fx:String>
        <fx:String>March</fx:String>
        <fx:String>April</fx:String>
        <fx:String>May</fx:String>
        <fx:String>June</fx:String>
        <fx:String>July</fx:String>
        <fx:String>August</fx:String>
        <fx:String>September</fx:String>
        <fx:String>October</fx:String>
        <fx:String>November</fx:String>
        <fx:String>December</fx:String>
      </s:ArrayList>
    </s:dataProvider>
  </s:DropDownList>
  <s:DropDownList selectedItem="@{orderInfo.cardExpirationYear}"
    requireSelection="true">
    <s:dataProvider>
      <s:ArrayList>
        <fx:String>2011</fx:String>
        <fx:String>2012</fx:String>
        <fx:String>2013</fx:String>
        <fx:String>2014</fx:String>
        <fx:String>2015</fx:String>
        <fx:String>2016</fx:String>
      </s:ArrayList>
    </s:dataProvider>
  </s:DropDownList>
</s:FormItem>
<s:FormItem>
  <s:Button label="Proceed" click="handleProceed( event )"/>
</s:FormItem>
</s:Form>
</s:Group>
```

- 11** Save `CreditCardInfo`. Open `CheckoutView` and add an instance of `CreditCardInfo` after the instance of `CustomerInfo`. Set the width and height to be 100%. Bind the `orderInfo` property to the `orderInfo` instance. Add an event listener for the `proceed` event that calls the `handleProceed` method, and passes the event object. Set the `includeIn` to be `billingInfo`.

```
<checkout:CreditCardInfo width="100%" height="100%"
  orderInfo="{orderInfo}"
  proceed="handleProceed( event )"
  includeIn="billingInfo"/>
```

- 12** Save the files, then open and run `FlexGrocer`. Now when you click checkout, you should be able to navigate from Page 1 to Page 2 of the checkout process.

Creating Review

There is just one page left to create for the checkout process: the page that will let the user see their information and choose to either complete the order or go back and make changes.

- 1** Right-click the checkout package, and choose new MXML Component. Leave the Package to be `views.checkout`, assign a name of `Review`, set the Layout to `spark.layouts.VerticalLayout`, and leave the Based on as `spark.components.Group`. Remove the Width and Height values. Click Finish.

- 2** Add a Script block with a public bindable property called `orderInfo` of type `OrderInfo`

```
<fx:Script>
  <![CDATA[
    import valueObjects.OrderInfo;
    [Bindable]
    public var orderInfo:OrderInfo;
  ]]>
</fx:Script>
```

- 3** After the Script block, create a label with the text “Checkout Page 3 of 3”.

```
<s:Label text="Checkout Page 3 of 3"/>
```

- 4** Below the label, add an `HGroup` with a width of 100% and a height of 90%

```
<s:Label text="Checkout Page 3 of 3"/>
<s:HGroup width="100%" height="90%">

</s:HGroup>
```

This `HGroup` will show the customer and billing information on the left and the shopping cart on the right.

- 5** As a first child of the `HGroup`, add a `Form` that contains a `FormHeading` with a label `Review and Checkout`. The form should also contain four `FormItem`s, with the labels `Name`, `Address`, and `Card Type`. The fourth `FormItem` should have no label.

```
<s:HGroup width="100%" height="90%">
  <s:Form>
    <s:FormHeading label="Review and Checkout"/>
    <s:FormItem label="Name">
    </s:FormItem>
    <s:FormItem label="Address">
    </s:FormItem>
    <s:FormItem label="Card Type">
    </s:FormItem>
    <s:FormItem>
    </s:FormItem>
  </s:Form>
</s:HGroup>
```

- 6** In the first `FormItem`, add a `Label`, whose text is bound to the `billingName` property of the `orderInfo` object.

```
<s:FormItem label="Name">
  <s:Label text="{orderInfo.billingName}"/>
</s:FormItem>
```

You don't need a two-way binding here, because the user is not able to edit the object.

- 7** In the second `FormItem`, add two labels. The first should be bound to `orderInfo.billingAddress`. The second will concatenate `orderInfo.billingCity` with a comma, space, `orderInfo.billingState`, space, and then `orderInfo.billingZip`.

```
<s:FormItem label="Address">
  <s:Label text="{orderInfo.billingAddress}"/>
  <s:Label text="{orderInfo.billingCity}, {orderInfo.billingState}
    {orderInfo.billingZip}"/>
</s:FormItem>
```

- 8** The third `FormItem` will have a label, whose text is showing the `orderInfo.cardType`.

```
<s:FormItem label="Card Type">
  <s:Label text="{orderInfo.cardType}"/>
</s:FormItem>
```

- 9** The final `FormItem` will have two buttons. The first should have a label of “Complete Order”, and the second label should read “Edit Information”. Add click handlers for both to call methods you’ll write shortly called `handleComplete` and `handleEdit`.

```
<s:FormItem>
  <s:Button label="Complete Order" click="handleComplete( event )"/>
  <s:Button label="Edit Information" click="handleEdit( event )"/>
</s:FormItem>
```

If you prefer, you can use quick assist (Ctrl+1) to create these methods for you, as you have done in previous exercises.

Next, you’ll add an instance of `CartGrid` to show the user the items in their shopping cart. But first you’ll need to have a shopping cart property that will contain the data for their cart.

- 10** In the Script block, just below the `orderInfo` property, create a bindable public property named `shoppingCart` of type `ShoppingCart`.

```
import cart.ShoppingCart;
import valueObjects.OrderInfo;
```

```
[Bindable]
public var orderInfo:OrderInfo;
```

```
[Bindable]
public var shoppingCart:ShoppingCart;
```

Don’t forget to ensure that `cart.ShoppingCart` is imported.

- 11** After the closing Form tag, add a VGroup with a width and height of 100%. Inside the VGroup, add an instance of your CartGrid component. Assign the CartGrid a width and height of 100%, and bind the shoppingCart property of the grid to the shoppingCart property of the Review class.

```
<s:VGroup width="100%" height="100%">
  <components:CartGrid width="100%" height="100%"
    dataProvider="{shoppingCart.items}"/>
</s:VGroup>
```

- 12** Listen for a removeProduct event from the cartGrid. Add an event handler that calls a function named removeProductHandler and pass it the event object as an argument. Use quick assist to generate the function for you.

```
<s:VGroup width="100%" height="100%">
  <components:CartGrid width="100%" height="100%"
    dataProvider="{shoppingCart.items}"
    removeProduct="removeProductHandler(event)"/>
</s:VGroup>
```

- 13** In the Script block, find the newly generated method. In the method, create a new instance of the ShoppingCartItem class, passing the event.product to its constructor. Then call the removeItem method of the shoppingCart and pass it your newly created ShoppingCartItem.

```
protected function removeProductHandler(event:ProductEvent):void
{
  var sci:ShoppingCartItem = new ShoppingCartItem( event.product );
  shoppingCart.removeItem( sci );
}
```

Be sure you add an import for the ShoppingCartItem class.

- 14** After the CartGrid, add a Label, whose text shows “Total:” followed by a binding to the shoppingCart.total property.

```
<s:VGroup width="100%" height="100%">
  <components:CartGrid width="100%" height="100%"
    dataProvider="{shoppingCart.items}"
    removeProduct="removeProductHandler(event)"/>
  <s:Label text="Total: {shoppingCart.total}"/>
</s:VGroup>
```

- 15** In the Script block, find the handleComplete method. Create and dispatch a new Event with the type completeOrder.

```
protected function handleComplete(event:MouseEvent):void
{
  dispatchEvent( new Event( 'completeOrder' ) );
}
```


- 16** In the Script block, find the `handleEdit` method. Create and dispatch a new Event with the type `editInformation`.

```
protected function handleEdit(event:MouseEvent):void
{
    dispatchEvent( new Event( 'editInformation' ) );
}
```

- 17** Create a Metadata block that defines the `completeOrder` and `editInformation` events.

```
<fx:Metadata>
    [Event(name="editInformation", type="flash.events.Event")]
    [Event(name="completeOrder", type="flash.events.Event")]
</fx:Metadata>
```

- 18** Save and close `Review.mxml`. This completes the `Review` component. Next you need to add it to the `CheckoutView`, and handle its events.

- 19** Open `CheckoutView.mxml`. In the Script block, add a public bindable property called `shoppingCart` of type `ShoppingCart`.

```
[Bindable]
public var shoppingCart:ShoppingCart;
```

- 20** Create an instance of the `Review` component below the `CreditCardInfo` component. Set the `width` and `height` to be 100%. Bind the `orderInfo` attribute to the `orderInfo` value object. Bind the `shoppingCart` attribute to the `shoppingCart` value object. Add event handlers for the `editInformation` and `completeOrder` events, and use `quick assist` to create those methods. Lastly, set the `includeIn` attribute to be `review`.

```
<checkout:Review width="100%" height="100%"
    orderInfo="{orderInfo}"
    shoppingCart="{shoppingCart}"
    editInformation="handleEdit( event )"
    completeOrder="handleComplete( event )"
    includeIn="review"/>
```

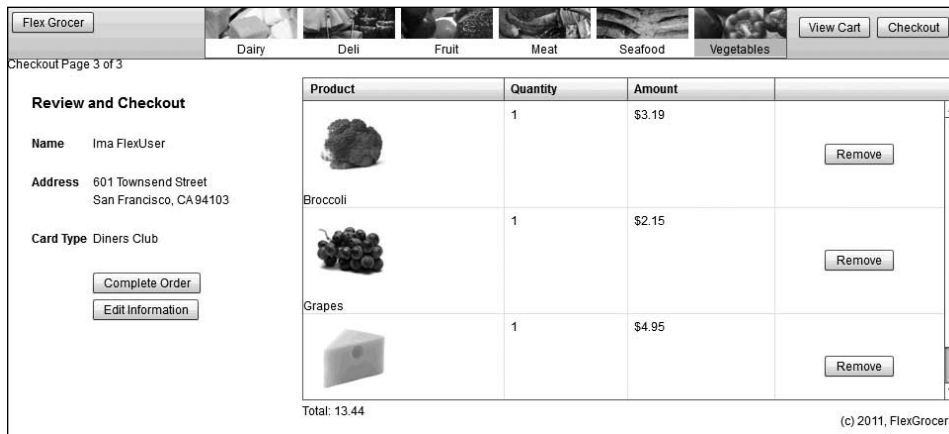
- 21** In the Script block, find the `handleEdit` method. In the body of the method, set the `currentView` to 0, and call `setViewByIndex`, passing in the `currentView`.

```
protected function handleEdit(event:Event):void
{
    currentView=0;
    setViewByIndex(currentView);
}
```

Earlier, you built the `currentView` and `setViewByIndex` to allow for easily switching between states. By telling the `CheckoutView` component to go to view 0, you are sending them back to the `CustomerInfo` screen.

- 22** Open FlexGrocer.mxml. Find the instantiation of the CheckoutView. Bind the `shoppingCart` property to `shoppingCart`.

```
<views:CheckoutView width="100%" height="100%"
    includeIn="checkout"
    shoppingCart="{shoppingCart}"/>
```



Completing the Order

The checkout process is almost complete. You still need an `OrderEvent` object to pass the `OrderInfo` back to the main application. You'll create that in this next exercise. For now, if you save the files and run the application, you'll see that you can add items to the cart, enter your information, step through to the confirmation page, and go back to edit the information.

- 1 Right-click the events package and choose new ActionScript Class. Leave the Package as **events**, set the Name to be **OrderEvent**, and add **flash.events.Event** as the Superclass. Click Finish.
- 2 In your new class, add a public property named `order` of type `OrderInfo`.

```
public var order:OrderInfo;
```

 Be sure to import the `valueObjects.OrderInfo` class.

- 3** Modify the constructor, so that an order object, of type `OrderInfo`, is passed as the second argument, between the `type` and `bubbles` parameters. In the body of the method, set `this.order` equal to the `order` parameter, just after the call to the super class.

```
public function OrderEvent(type:String, order:OrderInfo, bubbles:Boolean=false,
    ➤ cancelable:Boolean=false)
{
    super(type, bubbles, cancelable);
    this.order = order;
}
```

- 4** Override the `clone` method, so it returns a new instance of the `OrderEvent`. Your completed class should look like this:

```
package events
{
    import flash.events.Event;

    import valueObjects.OrderInfo;

    public class OrderEvent extends Event
    {
        public var order:OrderInfo;
        public function OrderEvent(type:String, order:OrderInfo,
            ➤ bubbles:Boolean=false, cancelable:Boolean=false)
        {
            super(type, bubbles, cancelable);
            this.order = order;
        }
        public override function clone():Event{
            return new OrderEvent(type, order, bubbles, cancelable);
        }
    }
}
```

- 5** Save and close the `OrderEvent` class.
- 6** Open `CheckoutView.mxml`. Find the `handleComplete` method. In this method, create and dispatch an `OrderEvent`, with the `type` set to `placeOrder`, and using the `orderInfo` object to populate its `order` property.

```
protected function handleComplete(event:Event):void
{
    dispatchEvent( new OrderEvent( 'placeOrder', orderInfo ) );
}
```

Remember that you'll need to import the `OrderEvent` class.

- 7** Still in the `handleComplete` method, reset the checkout process by setting the `currentView` to 0 and passing `currentView` to `setViewByIndex`. The final step of the method is to clear out the `orderInfo` property by setting it equal to a new `OrderInfo()` instance:

```
protected function handleComplete(event:Event):void
{
    dispatchEvent( new OrderEvent( 'placeOrder', orderInfo ) );

    currentView=0;
    setViewByIndex(currentView);

    orderInfo = new OrderInfo();
}
```

- 8** Add a Metadata tag block and declare the `placeOrder` event.

```
<fx:Metadata>
    [Event(name="placeOrder",type="events.OrderEvent")]
</fx:Metadata>
```

- 9** Save `CheckoutView`, and open `FlexGrocer`. Find the instantiation of the `CheckoutView`, and add an event handler for the `placeOrder` event, which will call a method named `handlePlaceOrder` and pass along the event object. Use the quick assist feature to have Flash Builder build this method for you.
- 10** Find the newly created `handlePlaceOrder` method. Inside the method, reset the shopping cart (by instantiating a new `ShoppingCart` in its place) and set the `currentState` to **shopping**:
- ```
protected function handlePlaceOrder(event:OrderEvent):void
{
 shoppingCart = new ShoppingCart();

 this.currentState="shopping";
}
```
- 11** Find the button with the **Flex Grocer** label, and add a click handler, which calls a method named `returnToShopping`. Pass an event object to the method. Use code assist to create this method.
- 12** In the newly created `returnToShopping` method, set the `currentState` to **shopping**:
- ```
protected function returnToShopping( event:MouseEvent ):void
{
    this.currentState="shopping";
}
```
- 13** Save and run the application.

The only remaining problem is that users can start the checkout process even if they have no items in their cart. Although this isn't really a problem, it can lead to user confusion.

In the final step, you'll dynamically enable/disable the checkout button, based on the contents of the cart.

- 14** Back in `FlexGrocer.mxml`, find the `btnCheckout` button and use a binding expression to enable the button only if the cart's total isn't \$0:

```
<s:Button id="btnCheckout" y="10" right="10" label="Checkout"  
➔ click="startCheckout(event)" enabled="{shoppingCart.total != 0}"/>
```

- 15** Save and run the application. This time it should be clearer to users that the checkout button is disabled until after they have added an item to the cart.

What You Have Learned

In this lesson, you have:

- Created a navigation structure to the application using states (pages 337–345)
- Used two-way binding to ease the process of sharing and editing data (pages 334–336)
- Implemented a full checkout process (pages 346–358)

This page intentionally left blank

LESSON 15

What You Will Learn

In this lesson, you will:

- Use a formatter and remove string concatenation
- Use a validator to check if data is in a valid format
- Learn to trigger validation in ActionScript

Approximate Time

This lesson takes approximately 1 hour to complete.

LESSON 15

Using Formatters and Validators

Flex provides built-in formatters and validators that enable you to display and validate user-supplied data such as dates, numbers, and currencies. Using the built-in data validators on the client side, you can make your application perform better by reducing calls to the server for validation. You can also save development time by using the built-in formatters to automate the often repetitive process of formatting data.

Customer Information

Customer Name ⚠ *This string is shorter than the minimum allowed length. This must be at least 2 characters long.*

Address

City

State

Zip *

Validating customer information

Introducing Formatters and Validators

Flex formatters convert raw data into a customized string using predefined rules. The formatters can be used in MXML or in ActionScript and work well with data binding to simplify the display of formatted data.

Validators are used to ensure that data meets specific criteria before the application attempts to use it. This can be particularly important if you expect a user to input a number for a mathematical operation or a date for scheduling. Like formatters, validators can be used in MXML or ActionScript. They provide logical feedback on data input (valid or invalid) but also provide visual feedback in the way of red borders and error messages when input is invalid.

Formatters

A formatter is simply an ActionScript class that implements an interface called `IFormatter`. Some of the formatters available include:

- `spark.formatters.NumberFormatter`
- `spark.formatters.CurrencyFormatter`
- `spark.formatters.DateTimeFormatter`
- `mx.formatters.CurrencyFormatter`
- `mx.formatters.DateFormatter`
- `mx.formatters.NumberFormatter`
- `mx.formatters.PhoneFormatter`
- `mx.formatters.ZipCodeFormatter`

You may notice that there are formatters (such as `Number` and `Currency`) that exist in both the `spark.formatter` package and the `mx.formatters` package. The formatters in the `spark.formatters` package are newer and have exciting new features such as locale awareness (the ability to format data differently depending on where the user lives/works/uses the application). So, whenever you have the choice, we recommend you use the latest and greatest from the `spark` package.

Formatters manage quite a bit of complexity for you, but they are exceedingly simple to use. Here is a `CurrencyFormatter` defined in MXML:

```
<s:CurrencyFormatter id="myFormatter"
    fractionalDigits="2"/>
```

This formatter can be applied either in ActionScript or in MXML with data binding, using the following syntax:

```
trace( myFormatter.format( 123.456789 ) );  
//outputs USD123.46 in the United States  
  
<s:Label text="{ myFormatter.format( someData ) }"/>
```

In the latter example, each time `someData` changes, the `format()` method will be called and the output displayed in the label.

Previously, you accomplished something similar using concatenation of strings. You wrote code like this:

```
<s:Label text="Your Cart Total: ${shoppingCart.total}"/>
```

This strategy has several problems. First, it becomes complicated to control variables defining how the user wants to see this currency presented. For example, if you are creating a globalized application, you'll need to support different currency symbols, different regional uses of commas and periods, and varying degrees of precision. Second, this code assumes that the currency symbol will always appear before the number. This is certainly not the case in many countries. By using Flex formatters, these and other issues are handled for you.

Validators

Flex also has a set of Validator classes that you can use to check whether a data type is valid (for example, if the input is a number) and to ensure that the input has been formatted correctly (for example, if a date is entered in a specific format). As with Formatter classes, you can use Validator classes either as MXML tags or instantiate them in ActionScript.

Using validators, you can perform a lot of data validation in the client application, instead of waiting until data is submitted to the server. Not only does this provide a more responsive user experience, it also reduces the number of calls between the client and the server. This yields a better-performing application. Client-side validation is not a perfect solution; some types of data validation (such as security issues) are still best performed at the server. But using Validator classes at the client reduces server calls to only these use cases.

All Flex validators implement an interface named `IValidator`. Some of the validators available as part of the Flex framework include:

- `spark.validators.NumberValidator`
- `spark.validators.CurrencyValidator`
- `mx.validators.CreditCardValidator`

- `mx.validators.DateValidator`
- `mx.validators.EmailValidator`
- `mx.validators.NumberValidator`
- `mx.validators.PhoneNumberValidator`
- `mx.validators.SocialSecurityValidator`
- `mx.validators.StringValidator`
- `mx.validators.ZipCodeValidator`

Much like the Formatter classes, the Validator classes cover a large number of use cases and conditions that you might not consider on your own. They are also split into two packages: `spark.validators` which contain the latest locale aware validators and the older `mx.validators`. Use the `spark.validators` if the particular item you need exists in both packages.

By using the Flex validators, you are better prepared for robust applications and internationalization requirements.

Using Formatter Classes

In this exercise, you'll apply a `CurrencyFormatter` class so all the price selections are displayed as local currency in the `FlexGrocer` application. There are multiple places in which prices are displayed in the application, including:

- The list of grocery products displayed
- The total of the shopping cart
- The subtotal and list prices in the user's shopping cart
- The checkout process

The `CurrencyFormatter` adjusts the decimal rounding, currency symbol type, and placement as well as sets the thousands separator and the negative sign.

1 Open `FlexGrocer.xml`.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer.fxp` project from the `Lesson15/start` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Within the `<s:Application>` tag, add a locale attribute and set the value to `en_US`.

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:views="views.*" xmlns:services="services.*"
  xmlns:cart="cart.*"
  creationComplete="handleCreationComplete(event)"
  locale="en_US">
```

By adding this attribute, you are specifying the locale of your application. A locale is a combination of a language and country code. So, in this case you are indicating that this particular application will be in English as spoken in the United States. Had you specified `en_GB`, the application would use English as spoken in Great Britain. Although English is spoken in both countries there are differences in spelling, units of currency, and so on. Later in this lesson, you'll specify a locale of `fr_FR`, French as spoken in France to see the differences applied to your application.

- 3 Open `ShoppingView.mxml` from your views package.
- 4 Within the `<fx:Declarations>` tags, add an `<s:CurrencyFormatter>` tag. Assign the tag an `id` of `currency` and add the `useCurrencySymbol` attribute with a value of `true`:

```
<fx:Declarations>
  <s:CurrencyFormatter id="currency"
    useCurrencySymbol="true"/>
</fx:Declarations>
```

- **TIP:** There is an `<mx:CurrencyFormatter/>` and an `<s:CurrencyFormatter/>` in this lesson. You are going to need to pay close attention to this difference to ensure the desired outcome.

The fractional digits, grouping separator, currency symbol, and many more properties can be set on the `CurrencyFormatter`; we have left these at their defaults. The `CurrencyFormatter` receives its default settings from user locale information. You also specified `useCurrencySymbol` of `true`, meaning that the formatter should include an appropriate currency symbol.

- * **NOTE:** If you would like to learn more about resource bundles or the process of internationalizing an application, refer to "Localization" in the Adobe Flex 4.5 help documentation.

- 5 Locate the Label control that displays the words *Your Cart Total* along with a dollar sign and the shopping cart's total. Inside the value for the `text` property, replace the dollar sign and binding expression with a call to the `format()` method of the currency object, and pass `shoppingCart.total` to the method, as follows:

```
<s:Label text="Your Cart Total{ currency.format( shoppingCart.total ) }"/>
```


- 11** Locate the function named `renderPriceLabel()`. Change the return statement of the function to use the `format()` method of the currency instance:

```
private function renderPriceLabel( item:ShoppingCartItem,
    column:DataGridColumn ):String {
    var subtotal:Number = item[ column.dataField ];
    return currency.format( subtotal );
}
```

Previously, you had to cast the subtotal as a `String` before concatenating with the `$`. This isn't necessary when using formatters. The `format()` method accepts an `Object` and internally converts it as needed.

- 12** Open `Review.mxml` from the `views/checkout` package.
- 13** Within the `<fx:Declarations>` tags, add an `<s:CurrencyFormatter>` tag. Assign the tag an `id` of `currency` and add the `useCurrencySymbol` attribute with a value of `true`.
- 14** Still inside the `<fx:Declarations>` tags, add an `<s:DateTimeFormatter>` tag. Assign the tag an `id` of `df`:

```
<fx:Declarations>
    <s:CurrencyFormatter id="currency"
        useCurrencySymbol="true"/>
    <s:DateTimeFormatter id="df"/>
</fx:Declarations>
```

- 15** Find the `FormHeading` tag for the Form displaying the Review and Checkout information. Add a new `FormItem` to the Form immediately after the heading with a `FormItem` label of `Order Date`:

```
<s:FormHeading label="Review and Checkout"/>
    <s:FormItem label="Order Date">
</s:FormItem>
```

- 16** Inside the `<fx:Script>` block, create a new private function named `getDate()`. This method will accept no arguments, and return a `Date` object.
- 17** Inside the `getDate()` method, instantiate and return a new `Date` object:

```
private function getDate():Date {
    return new Date();
}
```

You'll use this method to add the current date to the review page. You may remember that, in `ActionScript`, when a new `Date` object is created, it will reflect the current time.

- 18** Inside the Order Date form item, add a new `<s:Label/>` tag, set the text property equal to `df.format(getDate())`.

```
<s:FormHeading label="Review and Checkout"/>
<s:FormItem label="Order Date">
  <s:Label text="{df.format( getDate() )}"/>
</s:FormItem>
```

Like the `CurrencyFormatter`, the `DateTimeFormatter` has a `format` method used to convert data into a `String`. In this case, you are using the default format specified by the user's locale. You are passing the `format()` method the date returned from your `getDate()` method. By using the date object and the formatter, you are placing an indicator of the current date and time on the form.

- 19** Find the `Label` instance that displays the user's total. Pass the `shoppingCart.total` to the `format()` method of the currency instance before concatenating it and assigning it to the text:

```
<s:Label text="Total: { currency.format( shoppingCart.total ) }"/>
```

- 20** Save this file.

- 21** Open `ProductItem.mxml` from the components package.

- 22** Within the `<fx:Declarations>` tags, add an `<s:CurrencyFormatter>` tag. Assign the tag an `id` of `currency` and add the `useCurrencySymbol` attribute with a value of `true`.

- 23** Find the `Label` instance that displays the products `listPrice`. Pass the `product.listPrice` to the `format()` method of the currency instance before assigning it to the text.

```
<s:Label id="price" text="{ currency.format( product.listPrice ) }"/>
```

- 24** Save this file and run the application. If you add a few items to the cart and proceed through checkout, you should see formatted currencies and dates throughout.

Examining a Second Locale

In this exercise, you'll apply a different locale to the Application and then examine the results.

- 1** Open `FlexGrocer.mxml`.

- 2** Change the locale from `en_US` to `fr_FR` in the `Application` tag:

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:views="views.*" xmlns:services="services.*"
  creationComplete="handleCreationComplete(event)"
  locale="fr_FR">
```

3 Save and run the application.

Notice that the dollar signs have changed to euro symbols throughout the application. Further, the currency symbol now resides on the right side of the number.

4 Change the locale back to en_US and save the application.

Using Validator Classes

In this exercise, you'll use a `ZipCodeValidator` class to check whether a postal code is a valid U.S. zip code or Canadian postal code along with a `StringValidator` to ensure during the checkout process that the billing name is at least two characters long.

1 Open `CustomerInfo.mxml` from your `views/checkout` package.**2** Find the `FormItem` with the label `Customer Name` and set the `required` attribute of that `FormItem` to `true`.

```
<s:FormItem label="Customer Name" required="true">
  <s:TextInput text="@{orderInfo.billingName}"/>
</s:FormItem>
```

Setting the `required` attribute of a `FormItem` tag causes Flex to place an asterisk next to the form field when it is displayed. This is purely a visual property. By itself it does nothing to ensure the user enters data into this field.

3 Inside the `FormItem`, set the `id` property of the `TextInput` to `billingName`:

```
<s:FormItem label="Customer Name" required="true">
  <s:TextInput id="billingName" text="@{orderInfo.billingName}"/>
</s:FormItem>
```

Shortly, you'll need to refer to this `TextInput` by the `id` to validate its input.

4 Find the `FormItem` with the label `Zip` and set the `required` attribute of that `FormItem` to `true`:

```
<s:FormItem label="Zip" required="true">
```

Remember, this is purely a visual detail.

5 Inside the `FormItem`, set the `id` property of the `TextInput` to `billingZip`:

```
<s:FormItem label="Zip" required="true">
  <s:TextInput id="billingZip" text="@{orderInfo.billingZip}"/>
</s:FormItem>
```


- 6** Inside the `<fx:Declaration>` tag pair, add an `<mx:ZipCodeValidator>` tag. Bind the source property of the `ZipCodeValidator` to the `billingZip` `TextInput`. Still in the `ZipCodeValidator`, specify the property attribute as `text` and specify the `required` attribute to `true`:

```
<mx:ZipCodeValidator source="{billingZip}"
  property="text"
  required="true"/>
```

The `<mx:ZipCodeValidator>` validates that a string has the correct length for a five-digit zip code, a five-digit + four-digit U.S. zip code, or a Canadian postal code. The `source` attribute indicates the control containing the data to be validated. As you'll see, this also specifies where any error messages will appear. The `property` attribute indicates which property of the control you wish to validate. In this case you are indicating that the `text` property of the `billingZip` contains the data for validation. Finally, the `required` attribute indicates that this field must be supplied. If `required` was set to `false`, a blank field would be acceptable, but if the user entered any information, it must conform to a valid zip code.

If you used code completion when you added the `ZipCodeValidator`, the `mx` namespace was added for you at the top of this file. If you did not, be sure to add the `mx` namespace manually (`xmlns:mx="library://ns.adobe.com/flex/mx"`).

- 7** Still inside the `<fx:Declaration>` tag pair, add an `<mx:StringValidator>` tag. Bind the source property of the `StringValidator` to the `billingName` `TextInput`. Specify the property attribute as `text`, the `required` attribute as `true`, and `minLength` as `2`.

```
<mx:StringValidator source="{billingName}"
  property="text"
  required="true"
  minLength="2"/>
```

The `<mx:StringValidator>` validates that a string falls within certain size parameters. Here you are indicating that the `String` must be at least a length of `2` to be valid.

- 8** Save and compile the application.

Click the `Checkout` button and enter some letters for the zip code in the billing information screen. When you exit the field, you should see a red highlight around the text field. When you move the pointer over the text field, you'll see the default error message appear.

However, even if you leave these fields in error, you can still click the `Proceed` button to move on to the next screen. You'll correct that next.

- 9** Return to `CustomerInfo.mxml`.

You'll now add code to prevent leaving this page until the user corrects any errors.

- 10** Inside the `<fx:Declarations>` tag pair, wrap the two validators you created above in an `<fx:Array>` tag with the id of `validators`.

```
<fx:Declarations>
  <!-- Place non-visual elements (e.g., services, value objects) here -->
  <fx:Array id="validators">
    <mx:ZipCodeValidator source="{billingZip}"
      property="text"
      required="true"/>
    <mx:StringValidator source="{billingName}"
      property="text"
      required="true"
      minLength="2"/>
  </fx:Array>
</fx:Declarations>
```

This code creates an array named `validators`. It inserts the two validator instances created into that array so that they can be referred to as a group.

- 11** Find the `handleProceed()` method.

This method is called when the user clicks the `Proceed` button. It dispatches an event, which changes to the next view state.

- 12** Add a new local variable named `errors` of type `Array` on the first line of this method. Assign it to the result of calling the `Validator.validateAll()` method, passing it the `validators` array you just created.

```
var errors:Array = Validator.validateAll( validators );
```

If you used code completion, `mx.validators.Validator` will be imported for you. If not, import it now. The `validateAll()` method is a static method of the `Validator` class. It is a utility method that accepts an array of validators, like the one you created in step 10. It runs each validator and aggregates any failures, meaning that this array will contain any validation errors found as a result of running each of your validators. If the array is empty, there were no validation errors.

- 13** Just below the `errors` array declaration, create an `if` statement that checks if the `length` property of the `errors` array is `0`.

```
if ( errors.length == 0 ) {
}
```

Effectively, this `if` statement checks to see if there were any validation errors.

- 14** Move the code that dispatches the proceed event inside the `else` block.

```
if ( errors.length == 0 ) {  
    dispatchEvent( new Event( "proceed" ) );  
}
```

If there are no errors, the user will be allowed to continue.

- 15** Save and run the application. Enter invalid data in the Zip field and attempt to proceed to the next page.



The screenshot shows a form titled "Customer Information" with the following fields and a "Continue" button:

- Customer Name**: Input field with a red triangle icon and the text "This field is required." to its right.
- Address**: Input field.
- City**: Input field.
- State**: Input field.
- Zip**: Input field with a red triangle icon and the text "This field is required." to its right.
- Continue**: Button.

You now have a form capable of collecting valid data, informing users when that data is invalid, and preventing users from proceeding if they have not yet corrected the invalid data.

What You Have Learned

In this lesson, you have:

- Learned how to apply a formatter to incoming text (pages 362–368)
- Learned to set a locale for an application (pages 368–369)
- Learned how to apply a validator to outgoing data (pages 369–372)
- Learned to trigger validation from ActionScript (pages 371–372)

This page intentionally left blank

LESSON 16

What You Will Learn

In this lesson, you will:

- Learn how Flex applications are styled
- Set styles via tag attributes
- Learn about inheritable style properties
- Set styles via the `<fx:Style>` tag
- Set styles via CSS files

Approximate Time

This lesson takes approximately 1 hour to complete.

LESSON 16

Customizing a Flex Application with Styles

Out of the box, Flex provides a lot of functionality, but it has a rather generic look for an application. In this lesson, you'll explore how to apply basic customizations to a Flex application using styles applied both inline and via CSS style sheets.



The FlexGrocer application with a new font and highlight colors

Applying a Design with Styles and Skins

You can use one of two approaches to apply a design to your Flex applications: styles or skins. Styles allow you to modify the appearance of a Flex component by using style properties to set visual elements such as the font size and background color. In this lesson, you'll explore styles, learn about style inheritance, and learn about several ways to apply styles to your application.

Skins allow you to go beyond the functionality of styles, allowing you to change entire visual elements and rearrange those elements on the screen.

In previous versions of Flex, styles were the primary way applications were customized. In Flex 4 and later, designs for truly interesting user interfaces are a combination of the styling you'll learn in this lesson and the skinning techniques you'll learn in the next.

Cleaning Up the Appearance

Styling modifies the appearance of existing elements on the screen. So, before you begin styling the application, you'll make a few minor changes to the `ProductItem`'s appearance to make it more conducive to the final design.

- 1 Open `ProductItem.mxml` from your components package.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer.fxp` project from the `Lesson16/start` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Find the label with the `id` of `prodName` and move it directly below the `<s:Image/>` tag with the `id` of `img`.

Presently the product name is displayed above the image, but it will be moved lower for a cleaner appearance once styling is complete.

- 3 Wrap the `add` and `remove` `Button` instances in an `<HGroup>` tag pair with its `width` property set to `100%`.

```
<s:HGroup width="100%">
  <s:Button id="add" label="Add To Cart" click="addToCart( product )"/>
  <s:Button id="remove" label="Remove From Cart" click="removeFromCart(
product )"/>
</s:HGroup>
```

As opposed to being stacked vertically, the buttons will now be displayed side by side, providing more screen space for the products.

- 4 Shorten the label of the *Add To Cart* button to *Add*. Also shorten the label of the *Remove From Cart* button to *Remove*.

As these items are now side by side, shorter names will provide a cleaner look. Your final code should look like this:

```
<s:VGroup id="products">
  <s:Image id="img"
    scaleMode="letterbox"
    source="assets/{product.imageName}"
    mouseOver="this.currentState='expanded'"
    mouseOut="this.currentState='State1'"
    mouseDown="img_mouseDownHandler(event)"/>
  <s:Label id="prodName" text="{product.prodName}"/>
  <s:Label id="price" text="{ currency.format( product.listPrice ) }"/>
  <s:HGroup width="100%">
    <s:Button id="add" label="Add" click="addToCart( product )"/>
    <s:Button id="remove" label="Remove" click="removeFromCart( product )"/>
  </s:HGroup>
</s:VGroup>
```

- 5 Save the file and run the FlexGrocer application. The product name and buttons are now moved into a better position and ready for styling.

Applying Styles

As you have seen so far in your explorations, Flex development is performed using a number of standards-based languages, such as MXML (based on XML) and ActionScript 3.0 (based on ECMAScript). Styling is also accomplished in a standards-based way by using Cascading Style Sheets (CSS). You can apply a style by:

- Setting a single style on an individual component.
- Using CSS class selectors to set several styles together, which you can apply to multiple components.
- Using a type selector to specify that all components of a particular type (such as Button) should use a set of styles.
- Using descendant selection to indicate that components matching a particular hierarchy should use a set of styles (such as All Buttons in VGroup instances).
- Using an ID selector to specify that a component with a particular `id` should use a set of styles.
- Using pseudo-selectors, which allow you to style a particular state of a class (such as the Up state of a Button).

In the next several exercises, you'll have a chance to apply styles in all these ways.

Regardless of which way you apply a style, you need to know the style property that will affect the changes you want. The ASDocs, also known as the *Adobe Flex 4.5 Language Reference* (which ships with Flash Builder), have a complete list of all styles available for every built-in component in Flex.

For example, here are some common styles for the Label component:

- **color**: Color of text in the component, specified as a hexadecimal number.
- **fontFamily**: Name of the font to use, specified as a string, or a comma-separated list of font names. When you specify a list, Flash uses the first font found in the list. If you specify a generic font name (such as `_sans`), it will be converted to an appropriate device font. The default value is `Arial`.
- **fontSize**: Height of the text, specified in pixels. Legal values range from 1 to 720. The default value is 12.
- **fontStyle**: String indicating whether the text is italicized. Recognized values are `normal` (the default) and `italic`.
- **fontWeight**: String indicating whether the text is boldface. Recognized values are `normal` (the default) and `bold`.
- **paddingLeft**: Number of pixels between the container's left border and the left edge of its content area. The default value for Text controls is 0, but different defaults apply to other components.
- **paddingRight**: Number of pixels between the container's right border and the right edge of its content area. The default value for Text controls is 0, but different defaults apply to other components.
- **textAlign**: String indicating the alignment of text within its container or control. Recognized values are `left`, `right`, `center`, `justify`, `start`, or `end`. Flex 4 text controls support bidirectional languages such as Arabic and Hebrew, so the concepts of left and right can sometimes be a bit confusing. For a person reading left to right, left padding is at the beginning of the sentence. For a person reading right to left, right padding is at the beginning of the sentence. You can specify `start` or `end` for padding, and Flex will apply it to the left or right depending on the language. The default value is `start`.
- **textDecoration**: String indicating whether the text is underlined. Recognized values are `none` (the default) and `underline`.

This is just a small sampling of the styles available for text manipulation in Flex. Each component has its own list of style properties, such as the `selectionColor` or `rolloverColor` (used in components like `List` and `DataGrid`), which accept a color as a hexadecimal value to indicate the color of the background bar around an item when you either hover over or select it.

You can find a complete list of these styles in the ASDoc Help Files for each class.

ActionScript® 3.0 Reference for the Adobe® Flash® Platform
 Show Packages and Classes List | Packages | Classes | Index | Appendixes

Package and class filters: Runtimes: AIR 2.7 and earlier, Flash Player 10.3 and earlier, Flash Lite 4
 Products: Flex 4.5 and earlier, Flash Professional CS5.5 and earlier, LiveCycle Data Services 3.1 and earlier, LiveCycle ES2.5, BlazeDS 4, ColdFusion 9.0.1 and earlier, Open Source Media Framework 1.5 and earlier, Open Source Media Framework for Flex 4.0

spark.components
Label Properties | Methods | Events | Styles | Effects | Constants | Examples

focusBlendMode	Type: String CSS Inheritance: no Blend mode used by the focus rectangle.	UIComponent
focusSkin	Type: Class CSS Inheritance: no Skin used to draw the focus rectangle.	UIComponent
focusThickness	Type: Number Format: Length CSS Inheritance: no Thickness, in pixels, of the focus rectangle outline.	UIComponent
fontFamily	Type: String CSS Inheritance: yes The name of the font to use, or a comma-separated list of font names. For the Spark theme, see <code>flashx.textLayout.formats.ITextLayoutFormat.fontFamily</code>. For the Mobile theme, see <code>spark.components.supportClasses.StyleableTextField.Style.fontFamily</code>. The default value for the Spark theme is <code>Arial</code> . The default value for the Mobile theme is <code>_sans</code> . See also <code>flashx.textLayout.formats.ITextLayoutFormat.fontFamily</code> <code>spark.components.supportClasses.StyleableTextField.fontFamily</code>	Label
fontLookup	Type: String CSS Inheritance: yes Font lookup to use. For the Spark theme, see <code>flashx.textLayout.formats.ITextLayoutFormat.fontLookup</code> For the Mobile theme, this is not supported.	Label

Setting Styles Inline with Tag Attributes

You can apply styles to individual instances of a component by setting the tag attribute of the component with the name of the style property and the value you want to set. For example, to give a label a larger font size, specify the following:

```
<s:Label text="Only a Test" fontSize="40"/>
```

In this exercise, you'll set the `rolloverColor` and `selectionColor` for a `DropDownList` control in the second screen of the Checkout process (`CreditCardInfo.mxml`).

- 1 Open `CreditCardInfo.mxml` from your `/views/checkout` package that you used in the previous exercises.
- 2 Find the declaration for the first `DropDownList` control that displays credit card information. Add a tag attribute to specify the `rolloverColor` as `#AAAAAA`.

```

<s:DropDownList selectedItem="@{orderInfo.cardType}"
    requireSelection="true"
    rolloverColor="#AAAAAA">
  <s:dataProvider>
    <s:ArrayList>
      <fx:String>American Express</fx:String>
      <fx:String>Diners Club</fx:String>
      <fx:String>Discover</fx:String>
      <fx:String>MasterCard</fx:String>
      <fx:String>Visa</fx:String>
    </s:ArrayList>
  </s:dataProvider>
</s:DropDownList>

```

Letters used as part of a hexadecimal number (such as #AAAAAA) are not case sensitive; #aaaaaa works just as well.

- 3** Add another attribute to the same tag to specify the selectionColor as #EA800C.

```

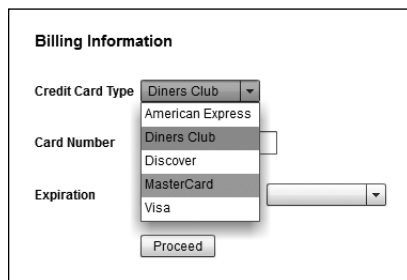
<s:DropDownList selectedItem="@{orderInfo.cardType}"
    requireSelection="true"
    rolloverColor="#AAAAAA"
    selectionColor="#EA800C">

```

You are now telling this DropDownList control that when a user hovers the pointer over one of the items, its color should be pale gray (#AAAAAA) instead of pale cyan (#0EFFF6), which is the default.

- 4** Save CreditCardInfo.mxml. Open and run FlexGrocer.mxml. Click Checkout in the upper-right corner. In the Customer Information form, fill out the required fields and click the Continue button. Click the Credit Card Type drop-down list and notice the color of selected and rolled-over items.

You can easily compare this with the default look of the DropDownList control because you have changed only one of the three controls on this screen. Open either of the other two drop-down lists to see the default selectionColor and rolloverColor.



► **TIP:** It is also possible to set styles on individual instances in ActionScript using the `setStyle()` method. For example, the same style could have been applied with this code:

```
idOfControl.setStyle("selectionColor",0xEA800C);
idOfControl.setStyle("rolloverColor",0xA8AAAA);
```

* **NOTE:** When using `setStyle()`, colors are prefixed with `0x`, which is the ECMAScript standard prefix for hexadecimal numbers. When applying a style in an attribute or `<fx:Style>` tag (as you'll soon see), you can use a pound sign (`#`) instead of `0x`. When set through ActionScript, numeric values (even those that are hexadecimal) do not have quotes around them.

Although `setStyle()` is useful for times when styles need to change at runtime, use it sparingly. `setStyle()` causes many of the visible portions of an application to refresh, so it is a processor-intensive operation.

Understanding Style Inheritance

As you look at the ASDoc for various components, you can see that each style has a yes or no property for something called CSS inheritance.

↑ chromeColor	Type: uint Format: Color CSS Inheritance: yes Theme: spark, mobile The main color for a component.	SkinnableComponent
↑ contentBackgroundAlpha	Type: Number CSS Inheritance: yes Theme: spark, mobile The alpha of the content background for this component.	List
↑ contentBackgroundColor	Type: uint Format: Color CSS Inheritance: yes Theme: spark, mobile The color of the content background for this component.	List
↑ cornerRadius	Type: Number Format: Length CSS Inheritance: no Theme: spark The radius of the corners for this component.	DropDownListBase
↑ dropShadowVisible	Type: Boolean CSS Inheritance: no Theme: spark Controls the visibility of the drop shadow for this component.	DropDownListBase
↑ focusAlpha	Type: Number CSS Inheritance: no Theme: spark, mobile The alpha of the focus ring for this component.	SkinnableDataContainer
↑ focusColor	Type: uint Format: Color CSS Inheritance: yes Theme: spark, mobile Color of focus ring when the component is in focus.	SkinnableDataContainer
↑ rolloverColor	Type: uint Format: Color CSS Inheritance: yes Theme: spark Color of the highlights when the mouse is over the component.	List
↑ selectionColor	Type: uint Format: Color CSS Inheritance: yes Theme: spark, mobile The color of the background of a renderer when the user selects it.	List

For example, in this figure you see that a few styles of the `DropDownList` control—`selectionColor` and `rolloverColor`—do allow CSS inheritance, whereas `cornerRadius` does not. What this means is that if a parent container of a `DropDownList` control has a value for `selectionColor` and the `DropDownList` control itself does not, the container's value will be used. However, because `cornerRadius` does not support inheritance, even if a parent container had a value set for `cornerRadius`, the `DropDownList` control would use the default value because it does not inherit this value.

Setting Styles with the <fx:Style> Tag

Many of you may have been exposed to CSS before when building web pages. You can also use many of the same CSS styles in your Flex applications. One way to do this is to add an <fx:Style> tag pair to the Application MXML document; you can write standard CSS style declarations between the open and close tags.

Standard CSS tends to have style properties whose names are all lowercase and uses hyphens as a separator between words:

```
background-color : #FFFFFF;
```

In the previous exercise, you used multiword styles by declaring them with *camel case* syntax; that is, the style declaration started with a lowercase letter and each subsequent word started with an uppercase letter, with no spaces or hyphens used:

```
<s:DropDownList rolloverColor="#AAAAAA"/>
```

The reason for the difference is that a hyphen is not a valid character for an XML attribute, and MXML tags are all XML tags. To work around this, when you set style names via attributes, set them with the ActionScript equivalent of the style name. So, for example, you use `backgroundColor` instead of `background-color`. The lowercase hyphenated versions of style properties are available only for properties that exist within traditional CSS. Any styles created specifically for Flex (such as `rolloverColor`) are available only in camel case. When you specify a style within an <fx:Style> tag, you can use either syntax, and Flex will apply it properly.

```
<fx:Style>
  .customDropDown{
    selection-color: #AAAAAA;
  }
</fx:Style>
```

or

```
<fx:Style>
  .customDropDown{
    selectionColor: #AAAAAA;
  }
</fx:Style>
```

Flex supports several ways to assign multiple styles at one time via CSS. These include class selectors, type (sometimes called element) selectors, descendant selectors, pseudo-selectors, and ID selectors.

Class Selectors

A class selector defines a set of style properties as a single style class, which can then be applied to one or more components through the use of the component's `styleName` property:

```
<fx:Style>
    .customDropDown {
        rolloverColor: #AAAAAA;
        selectionColor: #EA800C;
    }
</fx:Style>
<s:DropDownList styleName="customDropDown"/>
```

Here, the `DropDownList` control is using the `customDropDown` style class, which sets both the text `rolloverColor` and the `selectionColor`. You can use the `styleName` property to assign more than one style at a time to an instance by separating the style classes with a space:

```
<fx:Style>
    .customDropDown {
        rolloverColor: #AAAAAA;
        selectionColor: #EA800C;
    }

    .blueStyle {
        color: blue;
    }
</fx:Style>
<s:DropDownList styleName="customDropDown blueStyle"/>
```

In this case, the `DropDownList` control is using the `customDropDown` style class and the `blueStyle` style class, which sets the `rolloverColor`, `selectionColor`, and `color` of the text.

Type Selectors

A type selector enables you to specify a set of styles that will be applied to all instances of a component type. In HTML applications, you can do this to define the look of an `<H1>` tag for your site. The same basic syntactic structure works to define a set of styles to be applied to all instances of a Flex control type.

Throughout this book you have worked with namespaces. Flex uses namespaces as a means of clarification. This clarification allows you to specify the type of label you want, as in `<s:Label/>`, or perhaps which custom component you meant when indicating `<views:ShoppingView/>`. In ActionScript, you can have multiple classes with the same name but not in the same namespace. So, you could have a `Test` class in your `views` package and your `components` package, but you could not have two `Test` classes in the `views` package.

Namespaces allow you to be specific about the component you intend to address and ensure that the Flex compiler doesn't need to guess your intent.

The same concept is used when styling in CSS:

```
<fx:Style>
  @namespace s "library://ns.adobe.com/flex/spark";

  s|DropDownList {
    selectionColor: #EA800C;
    cornerRadius:5;
  }
</fx:Style>

<s:DropDownList id="stateProvenceCombo"/>
<s:DropDownList id="countryCombo"/>
```

In this example, the `cornerRadius` and `selectionColor` style properties are being applied to all `DropDownList` control instances in the Spark (`s`) namespace.

TIP: The terms type and class selector might seem counterintuitive if you haven't previously worked with CSS. These terms come from CSS standards, not from Adobe or Flex. The confusion is that a type selector is what you would use to affect all instances of an ActionScript class; a class selector has no relation to any ActionScript class, but instead defines a style class that can be used on several elements.

In this exercise, you'll build a class selector and apply it to an `<s:Form>` tag in `CreditCardInfo.mxml`. Not only will this showcase using a class selector, but you'll also see style inheritance in use as the style will be inherited by all the `DropDownList` controls in that form.

- 1 Open `FlexGrocer.mxml`.
- 2 Just after the closing `</fx:Script>` tag, add a new `<fx:Style>` tag pair.

When you add this tag, Flash Builder's code completion will take over and add a namespace for every namespace presently defined in the application. Your Style tag should look like the following:

```
<fx:Style>
  @namespace s "library://ns.adobe.com/flex/spark";
  @namespace views "views.*";
  @namespace services "services.*";
  @namespace cart "cart.*";

</fx:Style>
```

You now have an `<fx:Style>` block, in which you can create type or class selectors.

- 3 Inside the `<fx:Style>` block, create a class selector called `customDropDown` that specifies a `selectionColor` of `#EA800C` and a `rollOverColor` of `#AAAAAA`.

```
<fx:Style>
  @namespace s "library://ns.adobe.com/flex/spark";
  @namespace views "views.*";
  @namespace services "services.*";
  @namespace cart "cart.*";

  .customDropDown{
    selectionColor:#EA800C;
    rollOverColor:#AAAAAA;
  }
</fx:Style>
```

As with traditional CSS, but unlike style properties set as attributes, no quotes are used around the values of the style properties.

- 4 Open `CreditCardInfo.mxml`.
- 5 Remove the `rollOverColor` and `selectionColor` attributes of the `DropDownList` control. Instead, specify a `styleName` of `customDropDown` as an attribute on that `ComboBox` control.

```
<s:DropDownList selectedItem="{orderInfo.cardType}"
  requireSelection="true"
  styleName="customDropDown">
```

- 6 Save both `CreditCardInfo.mxml` and `FlexGrocer.mxml`, and then run the application.

The `DropDownList` instances in the Checkout section should behave exactly as they did before. The Credit Card Type will have custom colors, whereas the other two show the default colors.

- 7 Cut `styleName="customDropDown"` from the `DropDownList` and instead paste it as an attribute of the `<s:Form>` tag.

```
<s:Form styleName="customDropDown">
```

Because the form contains three `DropDownList` controls, applying these inheriting styles to the form will affect all the `DropDownList` controls within the form.

- 8 Save and run the application.

Verify that the style is now applied to all three `DropDownList` controls in the form.

Setting Styles with CSS Files

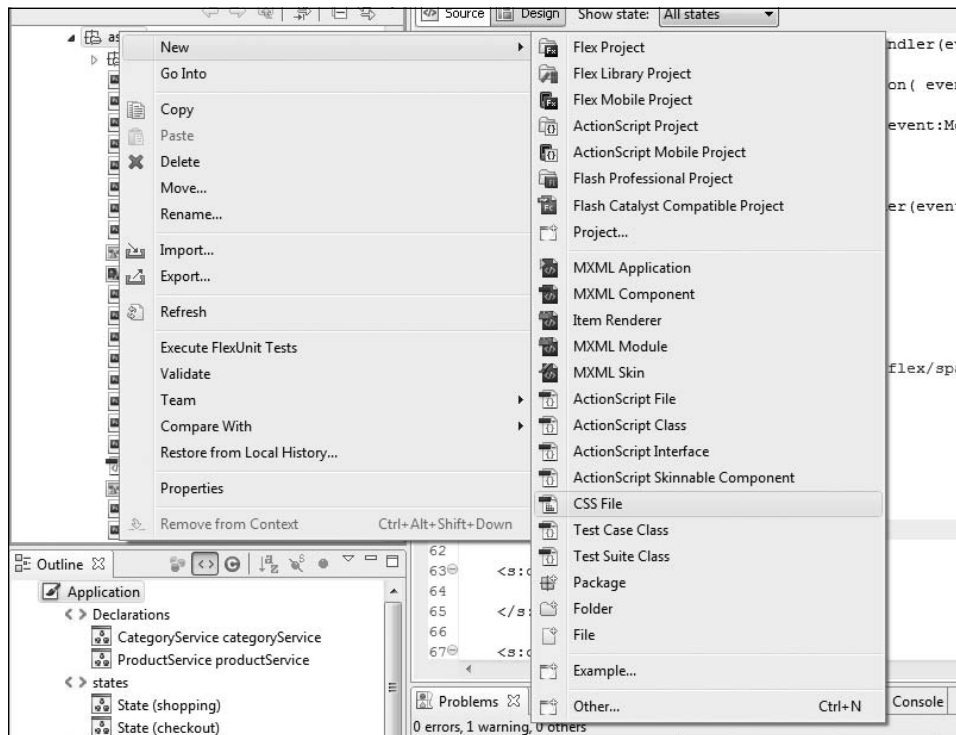
You can use an `<fx:Style>` tag to either define a block of styles inline on the MXML document, as you did in the previous exercise, or use its source attribute to specify an external CSS file to be compiled into the application.

```
<fx:Style source="path/to/file.css"/>
```

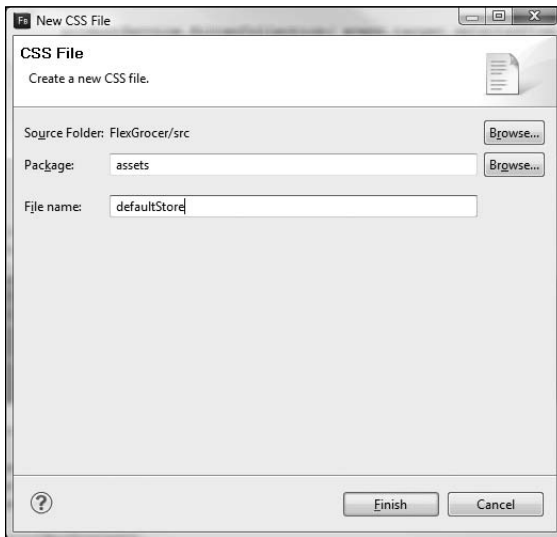
One great advantage of using an external file is that you can share CSS files between multiple Flex applications, or even between Flex and HTML applications. This is possible because CSS parsers in both Flex and HTML are smart enough to ignore any declarations they don't understand. So even if Flex supports only a subset of standard CSS, and in fact creates a number of its own custom declarations, neither your HTML nor your Flex applications will be hurt by declarations they cannot understand.

In this exercise, you'll create a CSS file and begin to style the FlexGrocer application.

- 1 Right-click the assets package of the Package Explorer. Choose **New > CSS File**.



- 2 Enter **defaultStore** as the name in the New File dialog box and click Finish.



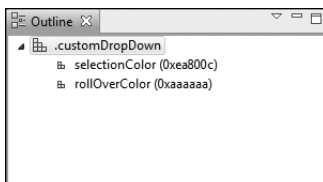
Flash Builder creates a new CSS File with the Spark namespace, ready for your customization.

- 3 Open FlexGrocer.mxml, find your `<fx:Style>` tag, and cut everything between the opening and closing tags. Paste this content into defaultStore.css. Your CSS file should have the following information:

```
@namespace s "library://ns.adobe.com/flex/spark";
@namespace views "views.*";
@namespace services "services.*";
@namespace cart "cart.*";

.customDropDown{
    selectionColor:#EA800C;
    rolloverColor:#AAAAAA;
}
```

Save this file. You might notice that the Outline view of Flash Builder understands CSS files as well as MXML and ActionScript. As your CSS becomes more complicated, the Outline view can be a great way to navigate through the file.



As a best practice, all styles for the application are defined in a single style sheet. This way, if you want to change the look and feel of the application at a later time, you don't need to dig through the code to find all the places where styles were applied; instead, you can restyle the application by changing only one file.

- 4 Return to `FlexGrocer.mxml` and find the `<fx:Style>` tag again. Convert the style tag from a tag pair to a single self-closing tag. Add a `source` attribute to the tag and sets its value to `assets/defaultStore.css`.

```
<fx:Style source="assets/defaultStore.css"/>
```

`FlexGrocer` will now use the external CSS file found in the `assets` directory for its style information.

- 5 Save `FlexGrocer.mxml` and run the application.

If all went as expected, the application will run and your `DropDownList` instances will still have custom coloring in the `CreditCardInfo` form.

Adding More Styling to the Application

You'll now have the opportunity to work with some of the other CSS selectors to apply styles to your application and components.

- 1 Open the `defaultStore.css` file you worked on in the previous exercise.
- 2 Just above the selector for the `customDropDown`, you'll embed a font for your `FlexGrocer` application using the CSS syntax. Do this by adding the following code:

```
@font-face {
    src: url("assets/fonts/SaccoVanzetti.ttf");
    fontFamily: SaccoVanzetti;
}
```

This code embeds the `SaccoVanzetti` font found in your `assets` folder. It associates that font with the `fontFamily SaccoVanzetti`, which you'll use to refer to this font elsewhere.

Embedding a font means the font is literally included in your application. This ensures that a user will be able to display the font exactly as you intended it to be seen—but it comes with a price. Just like embedding images or other assets, each time you embed a font, your application file size becomes larger.

The `SaccoVanzetti` font is part of the Open Font Library, which shares fonts under a Creative Commons License. Find more information about this font at <http://openfontlibrary.org/>

Although the font will now be included in your application, you have not specified where to use it.

- 3 Add a new type selector for the Application in the Spark namespace and specify that the Application class use the SaccoVanzetti font family.

```
s|Application {  
    fontFamily: SaccoVanzetti;  
}
```

This small bit of code includes several important concepts. First, you are indicating that you want to style the Application class in the Spark namespace. How do you know that? There are a few steps to unraveling this mystery.

First, notice that in your CSS file that there is a declaration on top for the Spark namespace. This line says you are going to use the letter `s` to represent the namespace found at the longer URI:

```
@namespace s "library://ns.adobe.com/flex/spark";
```

When you specify `s|Application` in your CSS file, you are clarifying that you mean the Application class found in the namespace is represented by the letter `s`.

If you were to look in your FlexGrocer application file, you would see a similar namespace declaration in the root tag:

```
xmlns:s="library://ns.adobe.com/flex/spark"
```

The difference in syntax is due to a difference in language. The `@namespace` declaration is how CSS defines namespaces. The `xmlns` declaration is how XML defines a namespace. The advantage of a standards-based language like Flex is a common set of ideas and languages that can be used between the web and your applications. The disadvantage of using all these standards is that, if you did not come from a background that uses all these discrete syntax elements, you sometimes need to learn several ways to say the same thing at the same time.

Ultimately, both namespaces are a way of referring to the same set of components. Because your FlexGrocer application begins with an `<s:Application>` tag, the small snippet of code that you added to your CSS file effectively indicates that you want to use this font for your main application.

Further, because the `fontFamily` is generally an inheriting style, by setting this style on your main application, you ensure that the same font will be used by all the controls in your project.

*** NOTE:** A note to readers with previous Flex experience: The newer text engine used in Spark components also uses a newer way to embed fonts. There is a check box in the Project Properties>Flex Compiler Options named 'Use Flash Text Engine in MX Components'. If you are working with your older Flex project and integrating some newer spark controls, this check box allows the older component to use the font you just embedded. The other option is to set a flag in the `@font-face` block called `embedAsCFF` to `false`. That flag will force the font to be embedded using the older method, which is natively compatible with MX components.

4 Ensure your CSS file looks like the following code, and then save and run the application.

```
@namespace s "library://ns.adobe.com/flex/spark";
@namespace views "views.*";
@namespace services "services.*";
@namespace cart "cart.*";

@font-face {
    src: url("assets/fonts/SaccoVanzetti.ttf");
    fontFamily: SaccoVanzetti;
}

s|Application {
    fontFamily: SaccoVanzetti;
}

.customDropDown{
    selectionColor:#EA800C;
    rolloverColor:#AAAAAA;
}
```

You should now see the SaccoVanzetti font applied to your application.

Using a Descendant Selector

In the previous exercise, you used a type selector to indicate that the entire application should be styled in a specific way. Using selectors will inevitably lead to conflicts when the same style is set two different ways; for example, when the font's color is set in one place to blue and in another place to black. In such a conflict, the most specific style wins. In other words, if you set the font's color to blue at an application level but set it to black on a label tag directly, the color for that label will be black as the label's attribute is more specific than the application setting.

Descendant selectors are a way to start adding specificity to your styling in place of generalities. Using descendant selectors, you specify the containment hierarchy as part of the styling. This means you can indicate that all classes of one type found inside all classes of another type should take on a specific style. The general syntax for a descendant selector is as follows:

```
ns|Component1 ns|Component2 ns|Component3 {  
  color: #FFFFFF;  
}
```

This particular style will only apply to instances of Component3, found inside Component2, found inside Component1. You can nest this hierarchy as deeply as you would like to maintain.

Here you'll choose to style all labels inside your ProductList component:

- 1 Open the defaultStore.css file you worked on in the previous exercise.
- 2 At the top of your CSS file under the existing namespaces, add a new one called components that maps to the components.* path.

```
@namespace components "components.*";
```

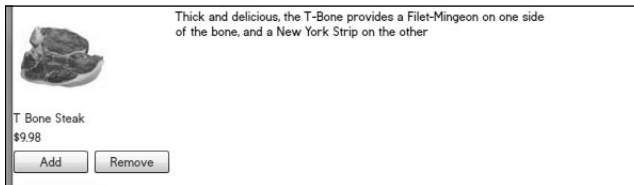
You'll use this namespace to apply styles specifically to classes that exist inside the components package.

- 3 Just above the selector for the customDropDown, add the following code:

```
components|ProductList s|Label {  
  color: #013FAC;  
}
```

In this case, you'll set the color style on all labels found inside the ProductList.

- 4 Save this file and run the application.



Note that the labels for any of your products, including name and price, are now blue. Because this application of styles is recursive, even the Add and Remove labels in the buttons inside the ProductList are blue. However, the description of the product is contained in a RichText tag, so it remains at its default color, along with the button for viewing the cart and similar buttons (since they were not inside the ProductList).

Using an ID Selector

So far you have applied styles to a specific component by creating a class selector and then using the `styleName` property on an MXML tag to apply that style. Using an ID selector is another approach that can be used when you wish to style just a single instance of a component.

Suppose you have a Label with an id of `myLabel`.

```
<s:Label id="myLabel" text="Hello"/>
```

You can apply a style to that instance by using a hash mark (#) combined with the id of the field:

```
#myLabel {
    color: #dfecdc;
}
```

This code will apply the color style to any control with an id of `myLabel`.

- 1 Open the `FlexGrocer.mxml` file.
- 2 Find the List instance in the `controlBarContent` that displays `NavigationItems`.
- 3 Add an `id` property to this List instance and set it to `categoryList`. Your List tag should look like the following:

```
<s:List id="categoryList" left="200" height="52"
    dataProvider="{categoryService.categories}"
    itemRenderer="components.NavigationItem"
    change="list1_changeHandler(event)">
    <s:layout>
        <s:HorizontalLayout/>
    </s:layout>
</s:List>
```

- 4 Open the `defaultStore.css` file.
- 5 At the bottom of the file, add the following ID selector for `categoryList`:

```
#categoryList {
    rolloverColor: #dfecdc;
    selectionColor: #6aa95f;
    borderVisible: false;
}
```

You are specifying new colors for both the selected and rollover colors for the list, as well as indicating that you no longer want to see any borders associated with this list.

- 6 Save this file and run the application.

If you choose an item from the List in the control bar, you'll now see different colors when hovering and when you select an item.

Using Pseudo or State Selectors

There is one remaining way to style components in Flex: using pseudo-selectors. With this approach, you style a view state of a component. For example, your main application has two view states (shopping and checkout and a Button has many (up, over, down, disabled, and so on). Using pseudo-selectors combined with any of the other techniques you have learned so far, you can style specific states of any Flex component.

The general form to apply a pseudo-selector in CSS looks like this:

```
ns|Type:viewState {
    color: #FFFFFF;
}
```

or this:

```
.class:viewState {
    color: #FFFFFF;
}
```

or this:

```
#id:viewState {
    color: #FFFFFF;
}
```

This code will apply the color style to any control with an id of myLabel.

- 1 Open the defaultStore.css file.
- 2 Just under the s|Application type selector, add a new type selector for the application, but specifically for the shopping state. Set the backgroundColor to #FFFFFF in this state.

```
s|Application:shopping {
    backgroundColor:#FFFFFF;
}
```

The application in the shopping state will be set to white.

- 3 Add another s|Application type selector specifically for the checkout state. Set the backgroundColor to #BBC8B8 in this state.

```
s|Application:checkout {
    backgroundColor:#BBC8B8;
}
```

The application in the checkout state will be set to a light green.

- 4** Next, below that customDropDown declaration, add a class selector named `cartButton` specifically for the over state. In this state, set the `chromeColor` style to `#F3FBF4`.

```
.cartButton:over {  
    chromeColor: #F3FBF4;  
}
```

- 5** Add another class selector for `cartButton` specifically for the down state. In this state, set the `chromeColor` style to `#C2CBE7`.

```
.cartButton:down {  
    chromeColor: #C2CBE7;  
}
```

You'll use these class selectors for every button dealing with cart navigation.

- 6** Open the `FlexGrocer.mxml` file.
- 7** Find the Button instance named `btnCartView`. Add a `styleName` property to the Button indicating it should use `cartButton` as its style.

```
<s:Button id="btnCartView" y="10" right="90" label="View Cart"  
    styleName="cartButton"  
    click="handleViewCartClick( event )"/>
```

- 8** Open the `ShoppingView.mxml` from the views package.
- 9** Find the Button instance with the label *View Cart*. Add a `styleName` property to the Button indicating it should use `cartButton` as its style.

```
<s:Button includeIn="State1"  
    height="17"  
    label="View Cart"  
    styleName="cartButton"  
    click="handleViewCartClick( event )"/>
```

- 10** Find the Button instance with the label *Continue Shopping*. Add a `styleName` property to the Button indicating it should use `cartButton` as its style.

```
<s:Button includeIn="cartView"  
    label="Continue Shopping"  
    styleName="cartButton"  
    click="this.currentState='"/>
```

- 11** Save any open files and run the application.

If you switch between the checkout view and shopping view, you should see a change in background color. If you hover over either of the View Cart buttons, you should see a different hover color and a different color again when you click on them.

Changing CSS at Runtime

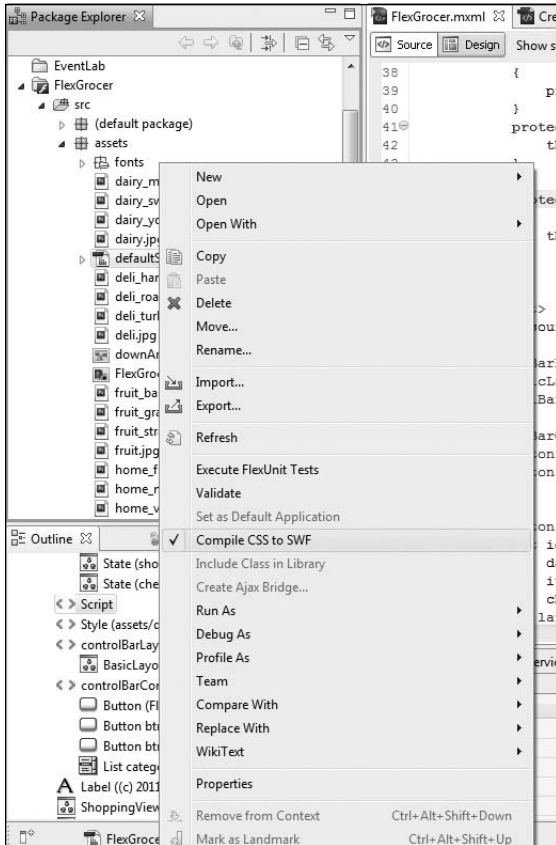
One drawback to the CSS approach shown in the previous section is that the CSS files are compiled into the application. This means that any changes to the application's style sheet require that the application be recompiled. A better approach is the ability to load CSS at runtime. You do not need to follow these steps at this time, but this section has been included for future reference.

Understanding the Benefits of Runtime CSS

There are a number of benefits to being able to change CSS at runtime. Chief among them is more rapid maintenance: A designer can simply deploy a new version of the CSS to the web server, eliminating the need to recompile and redeploy the application. Another benefit is a much easier approach for deploying a single application that can be presented with multiple skins, without the need for separately deployed applications for each skin. For example, if Flex Grocer wanted to partner with local grocery stores and allow the stores to brand the application as their own, it is now possible to have a single deployed version of the application, which loads a different style sheet depending on the domain from which the application has been loaded.

Creating a SWF from a CSS File

Flash Player doesn't natively have the ability to work with a runtime-loaded CSS file directly, so Adobe has added a simple mechanism for converting an existing CSS style sheet into a SWF, with which Flash Player can easily interact. Using the SDK, you can use the MXMLC compiler to compile a CSS file to a SWF, or it can be done even more easily within Flash Builder. All you need to do is right-click the CSS file in the Package Explorer and choose the Compile CSS to SWF option, as seen in the following figure.



Once the CSS has been compiled into a SWF, you can find the file named `defaultStore.swf` in your `bin-debug/assets` folder.

Loading a CSS SWF with StyleManager

Working with a CSS file compiled into a SWF is trivial; a single line of ActionScript is all you need to load and use that file. If you wanted to load your styles at runtime from the application, you would execute the following code from an event handler:

```
styleManager.loadStyleDeclarations("assets/defaultStore.swf");
```

This instructs StyleManager (an object in Flex responsible for managing all of the application's styles) to load the specified file and use any styles specified within it.

If you need to unload a CSS file loaded dynamically, there is another `StyleManager` method, `unloadStyleDeclaration`, that you'll find helpful:

```
styleManager.unloadStyleDeclaration("assets/defaultStore.swf");
```

Overriding Styles with a Loaded CSS

It's possible to have multiple style sheets in play. These can be a combination of compiled and dynamically loaded style sheets. The fundamental rule to remember when dealing with multiple style sheets is that if any styles are defined in more than one style sheet, the one loaded last is the one that Flex will use.

For example, if you have a CSS file compiled into the application with style definitions for `sApplication`, `.boldText`, and `.formHeading`, and you then load a CSS file at runtime that also has a definition for `sApplication` and `.formHeading`, the `.boldText` style from the compiled version will be used, as well as the `sApplication` and `.formHeading` style from the loaded style sheet—whichever is defined last is the one that Flex uses.

What You Have Learned

In this lesson, you have:

- Learned how Flex applications are styled (pages 376–379)
- Set styles via tag attributes (pages 379–381)
- Learned about inheritable style properties (page 381)
- Set styles via the `<fx:Style>` tag (pages 382–385)
- Set styles via CSS files (pages 386–394)
- Learned about runtime styling (pages 395–397)

LESSON 17

What You Will Learn

In this lesson, you will:

- Learn the relationship between skins and components
- Learn how to work with states and skins
- Create Button skins
- Create a skin for the application's controlBar region

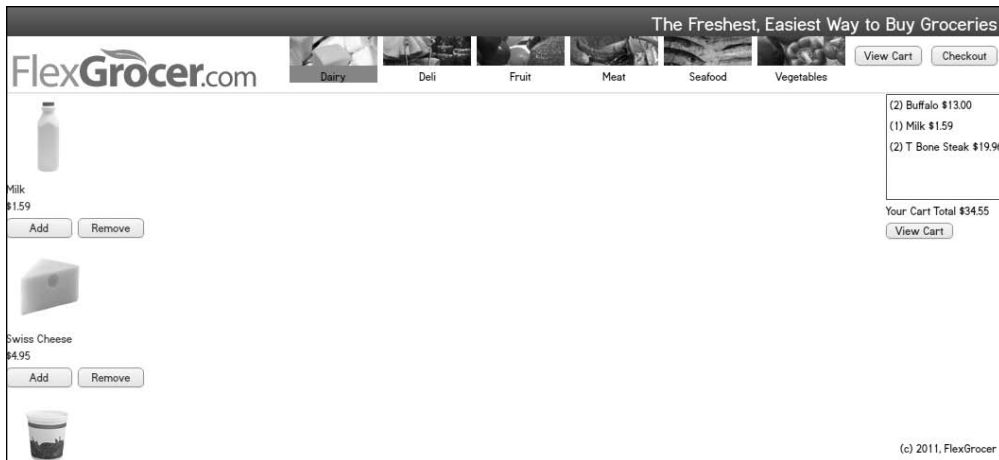
Approximate Time

This lesson takes approximately 2 hours to complete.

LESSON 17

Customizing a Flex Application with Skins

In the previous lesson, you learned about using the style API to customize parts of an application. You also learned that there are more customizations that you can make that are unavailable using the style API. In this lesson, you'll learn how to quickly and easily adjust the skins of a Spark component to completely change how that component looks.



The FlexGrocer.com application gets an extreme makeover through the use of a few simple skins.

Understanding the Role of Skins in a Spark Component

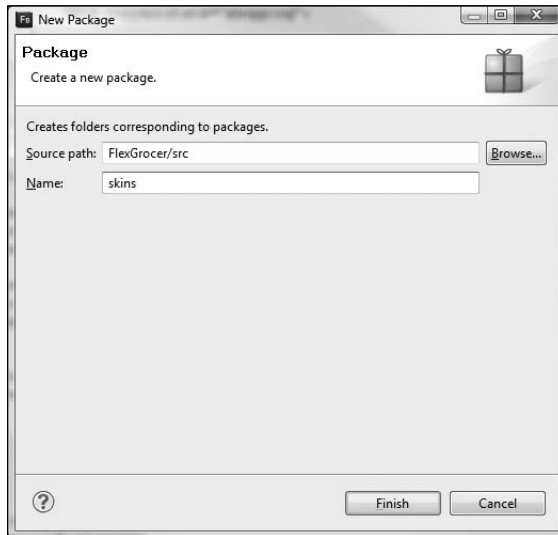
As you learned in Lesson 12, “Using the Flex DataGrid,” Spark components are built by composition, meaning that the functionality of the components is separated from the look of the component. In this lesson, you’ll learn how to adjust the look of the components through the use of skins.

In this exercise you’ll create a skin for the FlexGrocer button on the homepage. Up to this point, this button has simply had the text *FlexGrocer* on it. You’ll now modify the skin so it will display the FlexGrocer logo instead.

1 Open your FlexGrocer project.

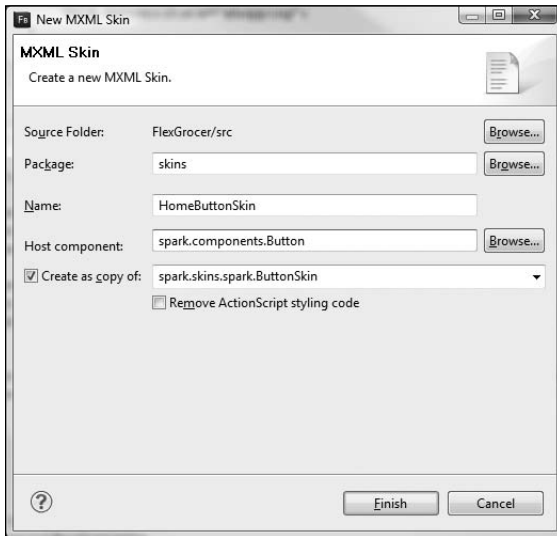
Alternatively, if you didn’t complete the previous lesson or your code is not functioning properly, you can import the FlexGrocer.fxp project from the Lesson17/start folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

2 Right-click the FlexGrocer project, and create a new package named **skins**.



This package will hold all the skin classes you create for the application.

- 3 Right-click the skins package, and choose **New > MXML Skin**. Name the skin **HomeButtonSkin**, set the Host component to **spark.components.Button**, choose to create it as a copy of **spark.skins.spark.ButtonSkin**, and then click **Finish**.



This will copy the native Spark ButtonSkin class and save it in your skins package under the name HomeButtonSkin. Skins must know the name of the class that they will be skinning, which allows the compiler to verify that all the proper pieces (known as skin parts) are present in the skin class. If any required skin parts are missing, a compile-time error will be thrown. If you have any questions on which skin parts are required for a given component, the ActionScript 3.0 Language references has a section for each component listing the skin parts, and whether or not they are required.

spark.components Button		
Properties Methods Events Styles Skin Parts Skin States Effects Constants Examples		
Skin Parts		
This component uses skins made up of skin parts. Do not set the skin parts directly. The component's skin sets the skin parts.		
Click for more information on skins ▼ Hide Inherited Skin Parts		
Skin Part	Description	Defined By
↑ iconDisplay:BitmapImage	Required: false Part Type: Static A skin part that defines an optional icon for the button.	ButtonBase
↑ labelDisplay:IDisplayText	Required: false Part Type: Static A skin part that defines the label of the button.	ButtonBase

As you can see in the figure, the Button has no required skin parts, which makes skinning a button easy to do.

- 4 Remove the Script block from the component, which was automatically added by the compiler.

The Script block in the component allows for programmatic control over the skinning, and lets you set some aspects of the skin in style sheets. Since you won't need this functionality for this skin, it is safe to remove the whole block.

- ▶ **TIP:** The new MXML skin dialog box has a check box named Remove ActionScript Styling Code. When selected it effectively deletes the Script block on your behalf.

- 5 Remove all the code between the end of the states block and the closing tag for this skin.

The resulting code for the skin class should look like this (comments from the top of the class have been intentionally omitted):

```
<s:SparkButtonSkin xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:fb="http://ns.adobe.com/flashbuilder/2009"
  minWidth="21" minHeight="21"
  alpha.disabled="0.5">

  <fx:Metadata>
    <![CDATA[
      /**
       * @copy spark.skins.spark.ApplicationSkin#hostComponent
       */
      [HostComponent("spark.components.Button")]
    ]]>
  </fx:Metadata>

  <!-- states -->
  <s:states>
    <s:State name="up" />
    <s:State name="over" />
    <s:State name="down" />
    <s:State name="disabled" />
  </s:states>

</s:SparkButtonSkin>
```

What you removed were the various elements that make up a label: the shadow, the fill color, a lowlight color, a highlight color, a highlight stroke, the border, and the label. For the moment, these are extraneous. Later in this lesson you'll add some of these elements back.

- 6** Between the end of the states block and the closing SparkButtonSkin tag, add a `BitmapImage` that uses an `Embed` directive for `assets/FlexGrocerButton.png` as its source. Specify a `horizontalCenter` of `0`, a `verticalCenter` of `1`, and the `alpha.disabled` in the disabled state as `.5`.

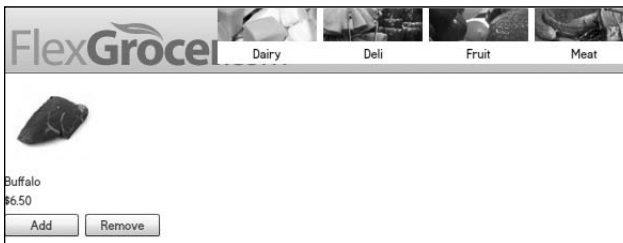
```
<s:BitmapImage source="@Embed('assets/FlexGrocerButton.png')"
  horizontalCenter="0" verticalCenter="1" alpha.disabled=".5"/>
```

You have now given this skin a visual look; instead of looking like a typical Flex Button, it will instead use this graphic as its complete look and feel.

- 7** Save `HomeButtonSkin.mxml`. Open `FlexGrocer.mxml` from the default package.
- 8** Find the instantiation of the button labeled Flex Grocer in the `controlBarContent` node. Add a `skinClass` attribute with a value of `skins.HomeButtonSkin`.

```
<s:Button label="Flex Grocer" x="5" y="5"
  click="returnToShopping( event )"
  skinClass="skins.HomeButtonSkin"/>
```

This code instructs this particular button to use the `HomeButtonSkin` class as its skin, instead of the class it would have used by default. If you save and run the application now, you'll find that the Flex Grocer button you have seen in the top-left corner has been replaced by a graphic, which still responds to users' clicks, just as the original button did.



- 9** Open `FlexGrocer.mxml`. Find the `List` with an `id` of `categoryList`. Remove `left="200"` and replace it with `right="171"`.

As the FlexGrocer button is now much larger than it was, the `List` component doesn't fit properly in the screen, so the category list will be constrained to stay 171 pixels from the right edge of the screen.

- 10** To prevent the category list from overlapping the logo if the browser is resized too small, set `minWidth="1024"` in the top `Application` tag.

```
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:views="views.*" xmlns:services="services.*" xmlns:cart="cart.*"
  creationComplete="handleCreationComplete(event)"
  locale="en_US"
  minWidth="1024">
```

One problem still persists with this button. It does not currently seem like a button because as users hover their mouse over it, or click it, it doesn't change its appearance at all, whereas all the other buttons in Flex do. You'll correct this problem in the next exercise, as you learn about states and how they relate to skins.

Understanding the Relationship between Skins and States

Early in this book, you learned how each Flex component can use different states to allow for adjusting and controlling how a component looks at various times throughout the application. States can also be modified in the component's skin as well.

Drawing Programmatically in Flex

The Flex framework offers tools for drawing programmatically: the AS3 Drawing API and FXG Graphics. In both cases, you can use code that instructs Flash Player how to draw graphical elements, rather than simply having it render a binary graphical file (such as JPEG, GIF, or PNG). The benefit of using a programmatic graphic is that it becomes much easier to control and change the graphic in your application than it would be when dealing with any of the binary formats. In most cases, if you wanted to change the color of a binary graphic, you would open that file in a graphics editing program, such as Fireworks or Photoshop, make the changes, and resave the file. When using programmatic graphics, you can simply adjust the properties of the graphical object that is drawing to the screen without having to be familiar with another program.

The AS3 Drawing API uses the `graphics` property that is native to all instances of the `Sprite` class in Flash Player. This API includes methods such as `moveTo`, `lineTo`, `beginFill`, `endFill`, and `curveTo`, all of which allow developers to draw vector graphics directly on a visual element in Flash Builder.

FXG, on the other hand, allows for XML-based syntax to define graphics, which work well directly inside MXML. In fact, if you were to open any of the Spark skin classes that ship with Flex 4, you would find that all the borders and background colors drawn in any of the Flex

components are done with a series of FXG declarations. Even better, many of Adobe's other tools, such as Photoshop, Illustrator, and Flash Catalyst, export their graphics as FXG, so you can use them directly in your Flex application.

The reality is you can do exactly the same thing with both FXG and the Flash Drawing API. Consider the following, which draws a similar red box with a blue border twice: first with FXG and then with the AS3 Drawing API:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  creationComplete="doDraw()">
  <fx:Script>
    <![CDATA[
      private function doDraw():void{
        var g:Graphics = drawingGroup.graphics;
        g.lineStyle(10,0x0000ff,.5);
        g.beginFill(0xff0000);
        g.drawRect(0,0,200,200);
        g.endFill();
      }
    ]]>
  </fx:Script>
  <s:Group x="100" y="10">
    <s:Rect width="200" height="200">
      <s:fill>
        <s:SolidColor color="0xFF0000" />
      </s:fill>
      <s:stroke>
        <s:SolidColorStroke weight="10" color="0x0000FF" alpha="0.5" />
      </s:stroke>
    </s:Rect>
  </s:Group>
  <s:Group id="drawingGroup" x="100" y="260"/>
</s:Application>
```



When this code runs, an identical box will be drawn twice, once using the Drawing API's `drawRect()` method and setting the `lineStyle` and `fill`, and the other time, using the FXG `Rect` tag, specifying `fill` and `stroke` as properties. The real benefit for Flex skinning in using FXG is that the XML markup can easily honor Flex states, so it would become trivial to change the look of the drawing as users move their mouse over the rectangle.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    creationComplete="doDraw()">
    <s:states>
        <s:State name="normal"/>
        <s:State name="over"/>
    </s:states>
    <fx:Script>
        <![CDATA[
            private function doDraw():void{
                var g:Graphics = drawingGroup.graphics;
                g.lineStyle(10,0x0000ff,.5);
                g.beginFill(0xff0000);
                g.drawRect(0,0,200,200);
                g.endFill();
            }
        ]]>
    </fx:Script>

    <s:Group x="100" y="10" mouseOver="this.currentState='over'"
        mouseOut="this.currentState='normal'">

        <s:Rect width="200" height="200">
            <s:fill>
                <s:SolidColor color="0xFF0000" color.over="0x00ff00" />
            </s:fill>
            <s:stroke>
                <s:SolidColorStroke weight="10" color="0x0000FF" alpha="0.5"
                    alpha.over="1" />
            </s:stroke>
        </s:Rect>
    </s:Group>

    <s:Group id="drawingGroup" x="100" y="260"/>
</s:Application>
```

In this example, two states are defined: a normal and an over state. The state is switched as you move the mouse over or off the rectangle. The fill and stroke of the rectangle are defined to change in the over state. This is how Flex achieves the different looks as the user interacts with the components in an application. If you were to open the `spark.skins.spark.ButtonSkin` class, you would see a series of rectangles defined that have fills or strokes, which change based on the state (up, over, down, or disabled).



For full details on the FXG specification, please see <http://opensource.adobe.com/wiki/display/flexsdk/FXG+1.0+Specification>.

Customizing Button States with Skins

In this next exercise, you'll continue to work with the Skin class you created for the homepage button in the previous exercise and you'll adjust how it looks in other states, such as when the user hovers the mouse over it or clicks on it.

- 1 Open `HomeButtonSkin.mxml`, which you created in the previous lesson.
- 2 Between the ending states tag and the Bitmap image tag, define a rectangle (use the Spark `Rect` class), which has an `id` of `fill`, and a value of 1 for the `top`, `bottom`, `left`, and `right` positioning.

```
<s:Rect id="fill" left="1" right="1" top="1" bottom="1">  
</s:Rect>
```

This block defines a `Rectangle` that will fill the component, except for 1 pixel on each of the four sides.

3 Add a tag pair to specify the fill property of the rectangle.

```
<s:Rect id="fill" left="1" right="1" top="1" bottom="1">
  <s:fill>
  </s:fill>
</s:Rect>
```

The fill property of the FXG shapes can accept any element implementing the IFill interface as a value. Among the framework classes that implement this interface are BitmapFill (which uses a graphic as a fill), LinearGradient (which will use a gradient of 2 or more colors as a fill along an axis), RadialGradient (which will use 2 or more colors as a fill, starting from a central point and radiating out to the edges), and SolidColor (which specifies a solid color to use as a fill). In the image here, you can see the four different fills native to the framework. Of course, the IFill interface is a relatively simple one, so you're free to build your own classes that implement it if the framework classes don't meet your needs.



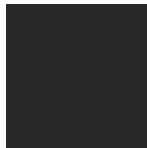
BitmapFill



LinearGradient



Radial Gradient



SolidColor

4 Populate the fill property with a LinearGradient, but add a LinearGradient tag as a child of the fill tag. Specify a rotation of 90 for the gradient.

```
<s:Rect id="fill" left="1" right="1" top="1" bottom="1">
  <s:fill>
    <s:LinearGradient rotation="90">
    </s:LinearGradient>
  </s:fill>
</s:Rect>
```

This creates an instance of the LinearGradient class and instructs the gradient to rotate 90 degrees from its standard top-to-bottom approach, so the gradient will appear left to right. All that remains is to instruct the gradient about the elements it will gradate between.

- 5** Create as children of the LinearGradient tag two instances of the GradientElement class. Both should have a color of white (0xffffffff); the differences will be the colors in the various states. The first Gradient should have an over color of white and a down color of olive green (0xafbcac). The second should have a light olive green (0xdfecdc) as both the over and down color.

```
<s:Rect id="fill" left="1" right="1" top="1" bottom="1">
  <s:fill>
    <s:LinearGradient rotation="90">
      <s:GradientEntry color="0xffffffff"
        color.over="0xffffffff"
        color.down="0xafbcac"
        alpha="1" />
      <s:GradientEntry color="0xffffffff"
        color.over="0xdfecdc"
        color.down="0xdfecdc"
        alpha="1" />
    </s:LinearGradient>
  </s:fill>
</s:Rect>
```

The ButtonSkin is told by the Button when the user is over it, or when the user presses down. You have taken advantage of that information by altering the color of your button's fill in these different states.

- 6** Save and run the application.

FlexGrocer.com
Up

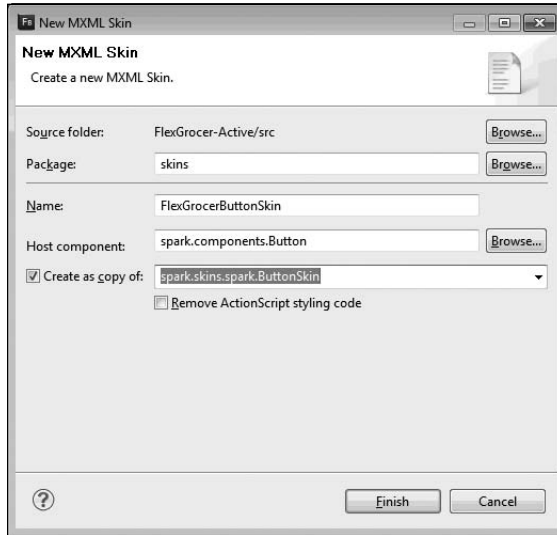
FlexGrocer.com
Over

FlexGrocer.com
Down

The Flex Grocer button now visually reacts to the users' gestures, giving them a much clearer indication that they can interact with it.

Now that you understand how to create a custom skin for a spark button, you are going to create another button skin; this one will be applied to all the other buttons in the application.

- 7 Right-click the skins package and choose New > MXML Skin.
- 8 Name the skin **FlexGrocerButtonSkin**, specify **spark.components.Button** as the Host component, and create it as a copy of **spark.skins.spark.ButtonSkin**.



- 9 In `FlexGrocerButtonSkin.mxml`, scroll down to the `<s:Rect>` tag with the id of `fill`. Find the first `GradientEntry` in the `LinearGradient`. Change the color to be pale green (`0xf3fbf4`), the over color to be a light olive green (`0xdfecdc`), and the down color to be a darker green (`0x6aa95f`). For the second `GradientEntry`, use a spring green as the color (`0xd4f1d8`) and a light olive green (`0xdfecdc`) for the over and down colors. Leave the default alpha of `.85` for both.

```

<s:Rect id="fill" left="1" right="1" top="1" bottom="1" radiusX="2">
  <s:fill>
    <s:LinearGradient rotation="90">
      <s:GradientEntry color="0xf3fbf4"
        color.over="0xdfecdc"
        color.down="0x6aa95f"
        alpha="0.85" />
      <s:GradientEntry color="0xd4f1d8"
        color.over="0xdfecdc"
        color.down="0xdfecdc"
        alpha="0.85" />
    </s:LinearGradient>
  </s:fill>
</s:Rect>

```

This will allow the buttons to use various greens, in keeping with the color palette of the application.

10 Delete the next five rectangles.

The design for the application's buttons do not require a lowlight, highlight, or highlight stroke.

- 11** The next Rect down has an `id` of `border`. Remove the child tags for the stroke from the Rect, and replace them with an instance of the `SolidColorStroke` class. Specify a color of the `SolidColorStroke` to be mint green (`0x8eb394`) with an `alpha` of 1 and a disabled `alpha` of .5.

```
<s:Rect id="border" left="0" right="0" top="0" bottom="0" width="69"
  height="20" radiusX="2">
  <s:stroke>
    <s:SolidColorStroke color="0x8eb394" alpha="1" alpha.disabled="0.5"/>
  </s:stroke>
</s:Rect>
```

Rather than using the complex linear gradient native to the `ButtonSkin` class, the `FlexGrocerButtonSkin` simply has a mint green border, which becomes more transparent when disabled.

There are a few references to the rectangles you removed in the class's `updateDisplayList()` method which must be removed or you'll encounter compile-time errors.

- 12** Scroll up to the `<fx:Script>` block and find the `updateDisplayList()` method.

- 13** Remove the lines that set the `radiusX` property of the `lowlight`, `highlight`, `highlightStroke`, `hldDownStroke1`, and `hldDownStroke2` rectangles. The revised `updateDisplayList()` should read like this:

```
override protected function updateDisplayList(unscaledWidth:Number,
unscaledHeight:Number) : void
{
  var cr:Number = getStyle("cornerRadius");

  if (cornerRadius != cr)
  {
    cornerRadius = cr;
    shadow.radiusX = cornerRadius;
    fill.radiusX = cornerRadius;
    border.radiusX = cornerRadius;
  }

  super.updateDisplayList(unscaledWidth, unscaledHeight);
}
```

As those rectangles are no longer in existence, you need to remove any references to them in the code.

The ButtonSkin is now complete; next you'll apply the skin to all Spark Buttons by using the CSS file.

- 14 Save and close FlexGrocerButtonSkin, and then open defaultStore.css from the assets directory.
- 15 After the end of the `s|Application:checkout` style declaration but before the start of the `ProductList Label` declaration, add a new Type Selector for the Spark Button class.

```
s|Application:checkout {
    backgroundColor:#BBC8B8;
}

s|Button {

}

components|ProductList s|Label {
    color: #013FAC;
}
```

- 16 Define a skin-class style property for the Button to use a class reference to the newly created `skins.FlexGrocerButtonSkin`.

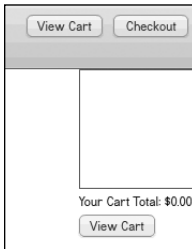
```
s|Button{
    skin-class:ClassReference('skins.FlexGrocerButtonSkin');
}
```

The `ClassReference` allows the CSS file to provide the `StyleManager` with a reference to your skin class. As you explore further in Flex, you'll find the `ClassReference` syntax is used any time you are providing an `ActionScript` class as a value for a style property.

- 17 Still in the Spark Button style declaration, specify a color (meaning font color) of `#1111b9` and a `corner-radius` of 5.

```
s|Button{
    skin-class:ClassReference('skins.FlexGrocerButtonSkin');
    color:#1111b9;
    corner-radius:5;
}
```

- 18** Save the CSS file and run the application. You should find the new look and feel in use throughout the application for buttons that don't already have their styles set more explicitly.

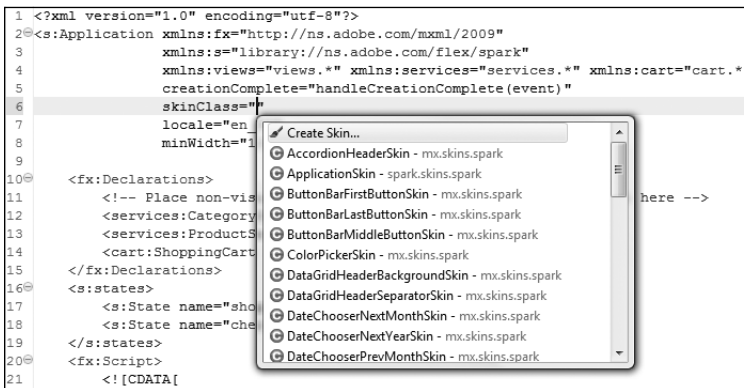


Creating a Skin for the Application

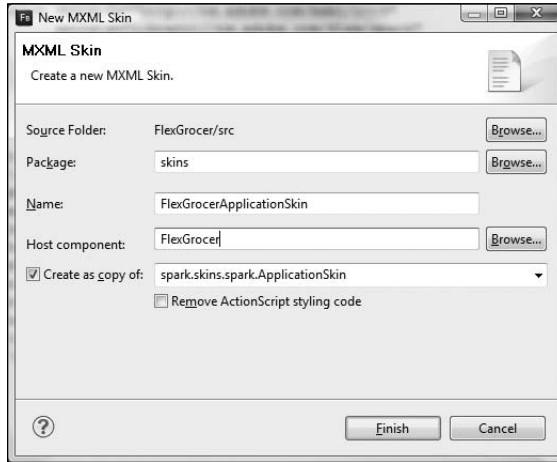
As you might imagine, skins don't apply only to buttons; larger, more complex components have skins as well. As mentioned earlier in the chapter, each Flex component has its look determined by the skins associated with it, and using the Spark components, you can easily customize any of them.

In this exercise, you'll create a skin for the Application component.

- 1** Open FlexGrocer.mxml. In the root `<s:Application>` tag, add a `skinClass` attribute. If you use the code-hinting feature, you'll find Flash Builder presents you with a list of skins or an option to create a new skin. (If you don't see the code-hinting, press Ctrl-Spacebar while your cursor is between the open and closing quotes of the attribute.) Choose the Create Skin option.



- 2** In the New MXML Skin dialog box, specify **skins** as the package, **FlexGrocerApplicationSkin** as the Name, and **FlexGrocer** as the Host component, and leave the other choices at their default values. Click Finish and save FlexGrocer.mxml.



Just like the other skins you created earlier in the lesson, this skin will be created in the skins package, and you'll start with the native Spark ApplicationSkin.

- 3** In the new FlexGrocerApplicationSkin file, find the Rect just below the comment that reads `<!-- layer 0: control bar highlight -->`. Remove the LinearGradientStroke being used as the stroke of that Rect, and remove its child tags. In its place add a SolidColorStroke with a light olive green color (0xdfecdc).

```
<!-- layer 0: control bar highlight -->
<s:Rect left="0" right="0" top="0" bottom="1" >
  <s:stroke>
    <s:SolidColorStroke color="0xdfecdc"/>
  </s:stroke>
</s:Rect>
```

For this application, the gradient stroke is not necessary; instead, a simple solid color stroke in the palette of the application will work fine.

- 4** Find the next `Rect` in the file, which will have the comment `<!-- layer 1: control bar fill -->` just above it. Change the `top` attribute from `1` to `32`.

```
<s:Rect left="1" right="1" top="32" bottom="2" >
  <s:fill>
    <s:LinearGradient rotation="90">
      <s:GradientEntry color="0xEDEDED" />
      <s:GradientEntry color="0xCDCDCD" />
    </s:LinearGradient>
  </s:fill>
</s:Rect>
```

You'll soon be adding a different rectangle above the control bar area, so here you are going to limit the background for the control bar area to start 32 pixels from the top of the application.

- 5** Still in the control bar fill `Rect`, remove the `LinearGradient` (and its child tags) used as the fill, and replace it with a `SolidColor` with a color value of white (`0xffffffff`).

```
<!-- layer 1: control bar fill -->
<s:Rect left="1" right="1" top="32" bottom="2" >
  <s:fill>
    <s:SolidColor color="0xffffffff"/>
  </s:fill>
</s:Rect>
```

Again, the native colors and gradients used as a background for the control bar group of a Flex application do not match the palette for the FlexGrocer application. Instead, a simple solid white background will work better.

- 6** Find the next `Rect` down, which has the comment `<!-- layer 2: control bar divider line -->` above it. Change the color of the `SolidColor` fill from black (`0x000000`) to light olive green (`0xdfecdc`).

```
<!-- layer 2: control bar divider line -->
<s:Rect left="0" right="0" bottom="0" height="1" alpha="0.55">
  <s:fill>
    <s:SolidColor color="0xdfecdc" />
  </s:fill>
</s:Rect>
```

You are once again replacing the default Flex colors with those matching your application.

- 7** After the end of the control bar divider line `Rect`, add a new `Rect`, with `x`, `y`, `left`, and `right` attributes set to `0`, and a height of `32`.

```
<s:Rect x="0" y="0" left="0" right="0" height="32">
</s:Rect>
```

This Rect will define a new area above the top navigation, which will house the company's motto.

- 8** Add a fill property to the rectangle. Populate the fill with a LinearGradient rotated 90 degrees. Add two GradientEntries to the LinearGradient. The first GradientEntry should have a Kelly green color (0x439235) with an alpha of 1. The second GradientEntry should have a dark Kelly green color (0x2E6224) with an alpha of 1.

```
<s:Rect x="0" y="0" left="0" right="0" height="32">
  <s:fill>
    <s:LinearGradient rotation="90">
      <s:GradientEntry color="0x439235" alpha="1"/>
      <s:GradientEntry color="0x2e6224" alpha="1"/>
    </s:LinearGradient>
  </s:fill>
</s:Rect>
```

The gradient fill in this rectangle will act as a background for the FlexGrocer tag line, which will appear at the top of the application.

- 9** Just below this rectangle, add a Label with the text **The Freshest, Easiest Way to Buy Groceries**. Set the top and right attributes to 10. Specify a color of white (0xffffffff) and a 20 point fontSize:

```
<s:Label text="The Freshest, Easiest Way to Buy Groceries"
  right="10" top="10"
  color="0xffffffff"
  fontSize="20" />
```

This label will be positioned in the top-right corner, on top of the gradient green background you created.

- 10** Find the Group under the comment `<!-- layer 3: control bar -->`. Adjust the top attribute to a value of 32.

```
<s:Group id="controlBarGroup"
  left="0" right="0"
  top="32" bottom="1"
  minWidth="0" minHeight="0">
```

By setting the top value to 32, you ensure the content of the controlGroup will be placed only over the white background, not over the green gradient.

- 11** Save FlexGrocerApplicationSkin. Open and run FlexGrocer.



What You Have Learned

In this lesson, you have:

- Learned about skins in a Spark component (pages 400–404)
- Learned about the relationship between skins and components (pages 404–407)
- Worked with states and skins (pages 407–413)
- Skinned the FlexGrocer application (pages 413–416)

LESSON 18

What You Will Learn

In this lesson, you will:

- Refactor code into an ActionScript component
- Create your own skin
- Manage skin parts and component life cycles
- Learn to use the Scroller

Approximate Time

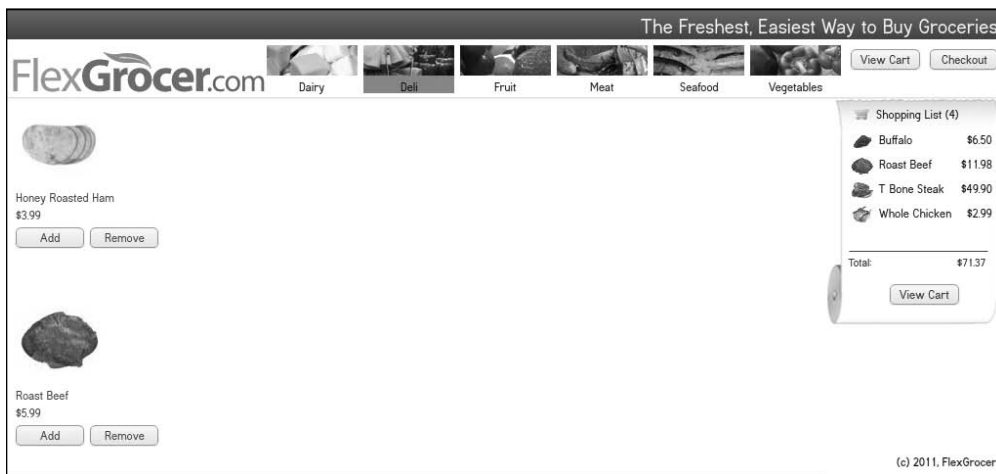
This lesson takes approximately 2 hours to complete.

LESSON 18

Creating Custom ActionScript Components

In Lesson 9, “Breaking the Application into Components,” you learned how to build custom components using MXML. There are times when you’ll need even more flexibility than MXML can offer. For these occasions, you can create components directly in ActionScript 3.0.

In this lesson, you’ll create a new component called `ShoppingList`, which will be an extensive refactoring of the existing `List` instance that shows your shopping cart items. It will include a new skin and new functionality, and will allow you to make a single component out of several separate pieces.



The FlexGrocer application with the new ShoppingList component

Introducing Components with ActionScript 3.0

In an earlier lesson, you learned that any code written in MXML is translated into ActionScript by the Flex compiler before being compiled into a SWF file. In reality, every Flex component is an ActionScript class, regardless of whether it's a UI control, a container, or some other type of component. Anything you might create in MXML can also be created in ActionScript, and there are things you can do with ActionScript that are not available purely from MXML.

The core Flex components you have used throughout this book—Label, DataGrid, and Button—are written in ActionScript. In general, components that are written for a single project or even quick prototypes of more advanced components are handled in MXML. However, if you want to build a very reusable and skinnable component, you'll eventually need to embrace ActionScript as your primary method.

The steps you'll take when creating an ActionScript 3.0 component are similar to the steps for building any ActionScript 3.0 class. First, determine what (if any) superclass your new class will extend. Then, determine what properties you need to declare for your class. Next, determine any new methods you might need to implement. You'll also need to declare any events your component will dispatch.

If your component is a visual class, as it will be in this lesson, you'll likely need to consider how your class will interact with a skin to allow you and others to change the visual appearance of your new component.

Building Components Can Be Complex

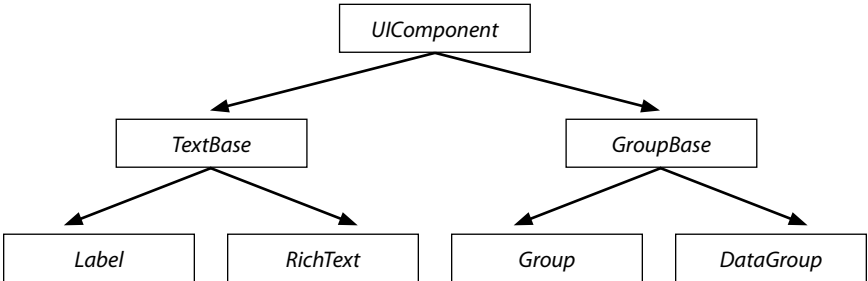
A word of warning: This lesson is the culmination of much of what you've learned in this book. Flex is intended to allow you to build applications quickly by assembling premade components. Flex can look easy, but it does so only because a component developer somewhere embraced the real complexity that lies just beneath the surface and wrestled it into submission.

When you develop Flex components, you become that individual, meaning that it is your job to make it look easy to the outside world by dealing with the complexity inside this little black box we call a component.

To create a well-developed component, you must balance the needs of your component's end user (sometimes you, sometimes your team or company, or at the most extreme, an unknown audience who will purchase and use it), with an understanding of Flash Player and the Flex framework. While this lesson will not be able to provide all of that understanding, it will touch on several areas of building a component.

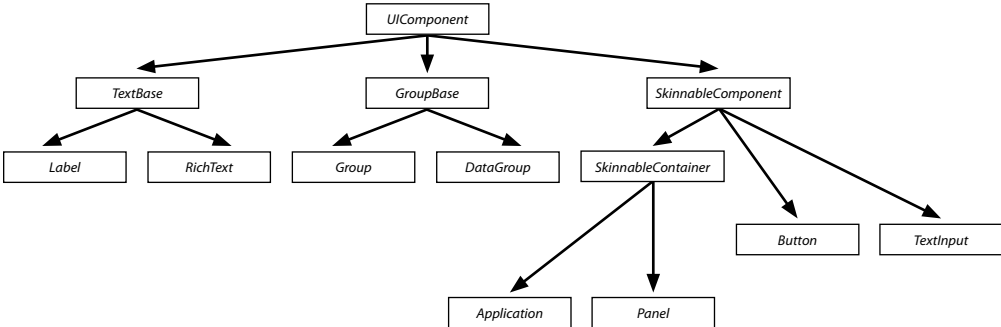
Understanding Flex Components

There are two types of components in the Flex framework: those that have a skinnable display and those that do not. Classes such as Group, DataGroup, and many others are not skinnable. That means you cannot apply the techniques learned in the previous lesson to change their appearance. These types of components are lighter weight and generally descend from a class named UIComponent.



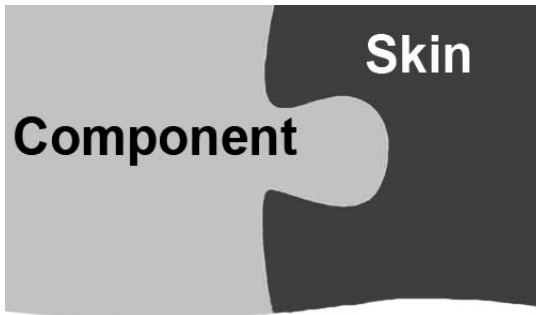
UIComponent is the base class for components in Flex and defines a lot of the functionality you have already become familiar with, such as automatic sizing and properties such as left, top, right, and bottom. Components that descend from UIComponent directly tend to be more self-contained with regard to their visual experience. In other words, they do not have a separate skin class that controls the way they appear on the screen.

Conversely, components may descend from SkinnableComponent. SkinnableComponent also descends from UIComponent, but it adds a key piece of additional functionality. It allows for the separation of all functionality related to the way the component appears to be defined in a separate class. Put simply, components that in some way descend from SkinnableComponent can have a skin.



Why does this all matter? It changes the definition of the word component. When using non-skinnable components such as Label, the component is mostly self-contained. Any intended visuals of the component must be contained within the class.

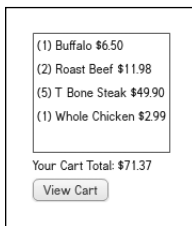
However, exactly the opposite is true of skinnable components. When creating a skinnable component, nothing about the way the component appears on the screen is defined inside the component itself; it is all defined in the skin. You can think of a skinnable component as consisting of two halves that must work together to create a whole.



Effectively, this is a separation of form and function. Skins are purely visual; they contain no logic for interactivity. Skinnable components do not have a visual representation without a skin; they are purely the functional aspects of a class.

Why Make Components?

Examining this image, you'll see your current shopping cart item view from your Flex Grocer application.



Right now this image does not represent a single component in your code: It is three separate components. This code from `ShoppingView` shows these declarations:

```
<:List id="cartList"
      dataProvider="{shoppingCart.items}"
      includeIn="State1"
      labelFunction="renderProductName"
      dragEnter="cartList_dragEnterHandler(event)"
      dragDrop="cartList_dragDropHandler(event)"/>
<:Label text="Your Cart Total { currency.format( shoppingCart.total ) }"/>
<:Button includeIn="State1" height="17"
        label="View Cart"
        styleName="cartButton"
        click="handleViewCartClick( event )"/>
```

Ignoring any visual changes for the moment, why might you want to make this into a single component as opposed to leaving it as is? The answer to that question comes down to the interface, encapsulation, and reuse.

To provide this display on the screen correctly right now, you need to remember to do the following:

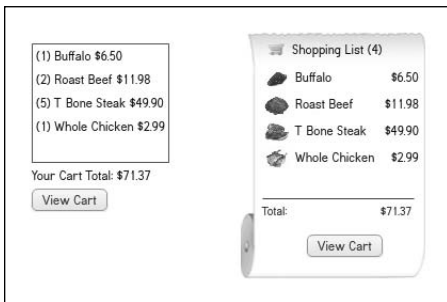
- Set the `includeIn` property correctly on a couple of different instances.
- Pass the shopping cart's items, not the entire shopping cart, to the `List` instance.
- Define a `CurrencyFormatter` on this page.
- Pass the total from the shopping cart into the `format()` function before passing it to the `Label`'s text property.
- Include functions to handle dragging and dropping.

While all those things are fine if you are the author of this code and only intend to duplicate this block once, consider it from another perspective. If you were going to instruct someone on how to add a listing of their shopping cart items to their page, would you want to explain each of those things? Suppose this component needed to be added in several different places in the application: Would copying all those pieces each time make sense? And lastly, in the abstract, all this code is currently in a class called `ShoppingView`. `ShoppingView`'s main job is to present a list of products and a view of the shopping cart to the user. Does code that understands how drag and drop works inside the `List` class really belong in `ShoppingView`?

The answer is no. When you create classes in an object-oriented language, you want to have a clear sense of what the resultant object will do. Ideally, it should have as singular a purpose as possible to keep the code maintainable and understandable. Right now `ShoppingView` does a variety of things and, from an object-oriented perspective, knows too much about the relationship of these objects to allow these pieces to be reusable. To solve that problem, you're going to take this one function that displays and handles the items in a user's shopping cart and refactor it into a new object with this purpose. Along the way, you'll gain the ability to skin this object and simplify its use—all because you have made the commitment to create a component from this functionality.

Defining a Component

You cannot create a component without knowing its intended purpose, so let's examine the current list and related controls alongside a new intended look and feel for this component.



Reexamining the code for the current implementation will give you an initial set of the requirements. You want to be able to replace this code and the associated functions in the `ShoppingView` with one component. Therefore, it must be able to do the same things.

```
<s:List id="cartList"
  dataProvider="{shoppingCart.items}"
  includeIn="State1"
  labelFunction="renderProductName"
  dragEnter="cartList_dragEnterHandler(event)"
  dragDrop="cartList_dragDropHandler(event)"/>
<s:Label text="Your Cart Total { currency.format( shoppingCart.total ) }"/>
<s:Button includeIn="State1" height="17"
  label="View Cart"
  styleName="cartButton"
  click="handleViewCartClick( event )"/>
```

Looking at this code, you should be able to gather a few important points. The component needs to:

- Display the contents of the `shoppingCart`'s items collection, which is just a collection of `ShoppingCartItem` instances.
- Accept drag-and-drop operations as a way of adding items.
- Display the shopping cart's total.
- Format the shopping cart's total.
- Facilitate switching to the `cartView` state.

This code uses generic Flex components: `List`, `Label`, and `Button`. Generic components are fantastic building blocks, but they force your application code, the code contained within files like `ShoppingView`, to do more work.

For example, you simply can't tell a generic component to display your `ShoppingCart` object. Instead, any person using this code has to provide the `List` with the items collection, the `Label` with the formatted total, and so on.

Generic Flex components aren't designed to understand concepts specific to your application. However, when you create your own custom components, you can tailor them to understand the objects that are important to your application and therefore reduce the pain in using them.

Defining the Interface

If you were to close your eyes and envision the perfect interface for your `ShoppingList`, what might that be? Perhaps instead of passing in items to a `List` and totals to a `Label`, you would just pass the entire `shoppingCart` and the component would know what to do with it. Perhaps instead of bothering you when someone dragged something onto the component or dropped it, the component would just tell you that there is a new product to add to the cart. This is an important exercise. When creating a new component one of the most critical things to get right is the interface—in other words, how the rest of the world will use your component in code. Here is the proposed interface for the new `ShoppingList` component:

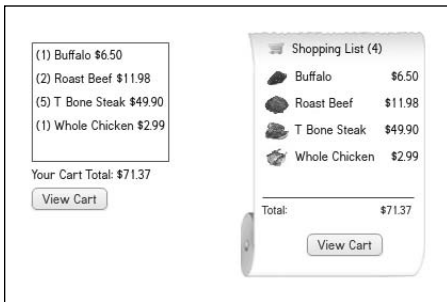
```
<components:ShoppingList
  shoppingCart="{shoppingCart}"
  addProduct="addProductHandler(event)"
  viewCart="currentState='cartView'"/>
```

The new component will accept a `ShoppingCart` instance as a property. It will let you know when a user attempts to add a product or clicks the view cart button via events that are easily handled. It will hide all the messy details internally, making it much easier to use—and reuse.

Choosing a Base Class

The last step before you begin creating your custom component is to choose a base class. That is the class you'll extend as your starting point. Choosing the base class for your new component is a critical decision and one you cannot make without a thorough understanding of the problem, so let's start there.

Reexamining the image from earlier, you'll see your current shopping cart item view on the left and the proposed shopping cart item view on the right. They look quite a bit different, but there are functional differences as well.



When you are deciding on a base class, you're trying to determine if there is another class that already does most of the needed work on your behalf. For instance, earlier you created `ProductList`. You did so by extending the `DataGroup` and changing a few things to make `ProductList` a viable component for your needs.

In this case, you are making a component that has an area to display a list of items. It also has an area to display the number of items in the cart, an area to display a total, and a `View Cart` button. Unlike `ProductList`, this component doesn't exactly mirror the functionality of any one Flex component. Instead, it's a composite of many different components interacting in a specific way.

As there isn't a component in Flex that provides you with the needed functionality, it will be up to you to build it all. While doing so, you are also going to allow for others in the future to change the way your component looks via skinning. Therefore, you'll use `SkinnableComponent` as your base class.

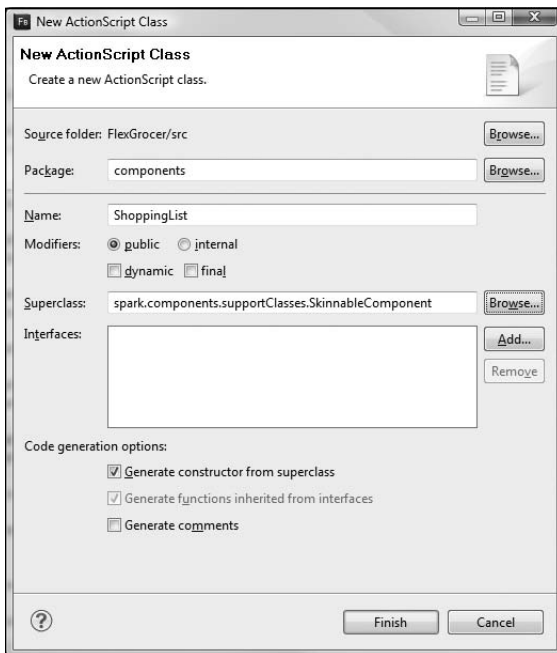
Creating the Class

You'll begin building the component to replace the shopping cart items list currently in `ShoppingView`. Start by creating a new `ActionScript` class.

- 1 Open the `FlexGrocer` project that you used in the previous lessons.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer.fxp` project from the `Lesson18/start` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Right-click the `components` package in the `Package Explorer`. Choose `New > ActionScript Class`.
- 3 Specify **ShoppingList** as the Name of the new class and `SkinnableComponent` as the superclass.



- 4 Click `Finish`.

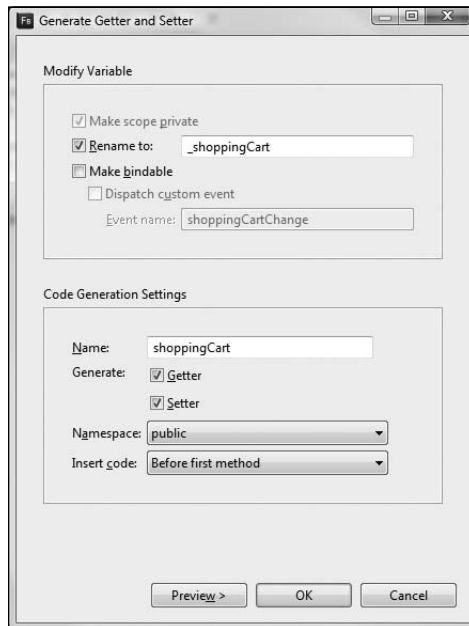
Now that you have a class, you need to define the interface explained earlier in code. The steps in this section rely heavily on Flash Builder. Learn to use these tools well, and you'll save immense amounts of time.

- 5** Above the constructor for the `ShoppingList` class, create a new private variable named `shoppingCart` of type `ShoppingCart`.

```
private var shoppingCart:ShoppingCart;
```

Be sure to use code completion when typing so that Flash Builder imports `cart`. `ShoppingCart` on your behalf.

- 6** Right-click `shoppingCart` in the line of code you just wrote. From the pop-up menu, choose `Source > Generate Getter/Setter`.
- 7** The Generate Getter/Setter dialog box opens. Ensure your options look the same as those in the following image:



This dialog box will generate a new getter and setter function on your behalf, saving you typing and typos.

- 8** Click OK. You should now have a getter and setter function for a `shoppingCart` property, and your original variable will be renamed with an underscore.

```
private var _shoppingCart:ShoppingCart;

public function get shoppingCart():ShoppingCart {
    return _shoppingCart;
}

public function set shoppingCart(value:ShoppingCart):void {
    _shoppingCart = value;
}
```

This property was the first of three things that made up the `ShoppingList` interface. The remaining two are both events, which you'll add next.

- 9** Move to just above the `ShoppingList` class definition and add event metadata for an event named `addProduct` that will dispatch an event of type `events.ProductEvent`.

```
[Event(name="addProduct", type="events.ProductEvent")]
```

- 10** Add another piece of event metadata just below the last for an event named `viewCart`, which will dispatch an event of type `flash.events.Event`.

```
[Event(name="viewCart", type="flash.events.Event")]
```

- 11** Manually import the two event classes at the top of your file.

```
import events.ProductEvent;
import flash.events.Event;
```

- 12** Save this file.

Your public interface is now complete, and you can change your MXML to use your new component.

Using Your Custom Class

Although your new component does not yet have any functionality useful to the user, its public interface is complete. This means you can replace your existing code with this new component.

This is a great way to check your design and ensure you met all the requirements before continuing with implementation. If your component can be dropped into the place where it is eventually needed, you likely have the basics covered.

- 1** Open the `ShoppingView` from the `views` package.
- 2** Find the `VGroup` named `cartGroup` that contains the components responsible for showing the cart's contents in different views.

- 3** Delete the List control, the Label that displays the cart total, and the Button that is responsible for switching to the cartView state. Your code for this VGroup should look like this:

```
<s:VGroup id="cartGroup" height="100%"
    width.cartView="100%">
    <components:CartGrid id="dgCart"
        includeIn="cartView"
        width="100%" height="100%"
        dataProvider="{shoppingCart.items}"
        removeProduct="removeProductHandler(event)"
    />

    <s:Button includeIn="cartView"
        label="Continue Shopping"
        styleName="cartButton"
        click="this.currentState=' '"/>
</s:VGroup>
```

- 4** Next, add your ShoppingList component just above the dgCart but still inside cartGroup and pass it a reference to the shoppingCart. Previously, the List was only included in State1, so also add that logic to this tag.

```
<components:ShoppingList
    includeIn="State1"
    shoppingCart="{shoppingCart}"/>
```

- 5** Now handle the addProduct event by calling the addProductHandler event listener, which is already defined in this view.

```
<components:ShoppingList
    includeIn="State1"
    shoppingCart="{shoppingCart}"
    addProduct="addProductHandler(event)" />
```

Technically this component already has a reference to the shoppingCart, which means you could manually add a new product anytime you wanted without dispatching and handling this event. However, there are two good reasons not to do so. First, there is already logic on this view to handle the Add Product button click from the ProductList. Reusing this logic means less duplication, but more importantly it means if this logic needs to change, it changes in only one place.

Further, while you are making this component more specific, it is still best to separate the logic that your application performs from the way it is displayed. This component is about displaying items in a specific way and interacting with the user. You really don't want it to also have the responsibility of understanding how products are added to the

cart or you're back to having components that know too much—part of what we're correcting by moving some of this code out of ShoppingView.

- 6 Handle the `viewCart` event by setting `currentState` to `cartView`. The final tag should look like this:

```
<components:ShoppingList
    includeIn="State1"
    shoppingCart="{shoppingCart}"
    addProduct="addProductHandler(event)"
    viewCart="currentState='cartView'"/>
```

Your new component is now taking the place of the older pieces, but there is now extraneous code in `ShoppingView` that can be eliminated—the functionality will be moved into the `ShoppingList` component.

- 7 Delete the `renderProductName()`, `cartList_dragEnterHandler()`, and `cartList_dragDropHandler()` methods from `ShoppingView`. You may also delete the following imports, which were used only by these methods:

```
import mx.core.UIComponent;
import mx.events.DragEvent;
import mx.managers.DragManager;
import mx.core.DragSource;
```

The functionality of these methods belongs to the `ShoppingList` now and will no longer be needed in `ShoppingView`.

- 8 Save all your files. You shouldn't see any compilation errors, but if you were to run this code now you'd receive an error at runtime.



You presently have function with no form. You've learned that components based on `SkinnableControl` are really two halves, one side representing the function and the other the form. Flex can't figure out what you want displayed on the screen. You'll deal with that issue next.

Creating the Visuals

You created the stub for your new custom component in the previous section, but now you want to define its visual appearance and then link the two together. Defining the requirements for these two components to talk and establishing the visual display will be the focus of this section.

Specifying the Skin Requirements

Components that support skinning in Flex are composed of two pieces. This separation provides enormous capability but also some complexity. The two halves need to communicate and they need to set requirements for each other. The functional side of the component in your case will be responsible for displaying the total. Therefore, it needs to know that there will be a label created by the visual side allowing that to happen.

These requirements are set via three metadata tags that collectively help tame the madness of this dynamic component model. You learned about these tags briefly in Lesson 17, “Customizing a Flex Application with Skins”; however, you’ll now use them to define your component.

The first metadata tag is called `SkinPart`. The `SkinPart` metadata is responsible for defining what pieces are required of the skin to be considered legitimate. Using your component as an example, the `ShoppingList` will need to indicate that it needs some place to put the total, the quantity, and the items. The Flash Builder environment will not allow someone to assign a skin to your component that does not implement all these required parts.

The `SkinPart` metadata is used inside the class and above a property. In this example:

```
[SkinPart(required="true")]  
public var myLabel:Label;
```

a component is indicating that the skin must have a `Label` named `myLabel` to be considered a valid skin. If `required` is set to `false`, it is optional for the skin to implement.

The next piece of metadata is called `SkinState`. The `SkinState` metadata tag is responsible for indicating what states are required of the skin. The simplest example of this is the normal and disabled state. In Flex you can set the `enabled` property for any `UIComponent` to `false`. Doing so should prevent interaction with the component and often changes the component visually to ensure the user perceives the reason for the lack of interaction.

```
[SkinState("normal")]  
[SkinState("disabled")]
```

When this metadata is added above the class declaration for a component, it means that any skin for this component must have these two states defined. It does not prescribe what the

skin does during a state change. For instance, it is completely your choice if the skin blinks or does nothing in a disabled state, but it must be able to handle this state change in whatever way you see fit.

The final piece of metadata important to skinning resides in the skin itself. This piece of metadata is called `HostComponent`.

```
[HostComponent("components.MyList")]
```

The `HostComponent` tag is used to associate a skin with its component. In other words, it is used to indicate which halves make the whole. This is extremely important as it allows Flash Builder to do compile-time checking on your behalf. If you create a new skin and specify that it is for a particular component, Flash Builder can check the `SkinState` and `SkinPart` metadata of the named component and verify that your skin meets those requirements. That way, you know at compile time, instead of runtime, if there is a problem.

- 1 Open the `ShoppingList.as` file that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer-PreSkin.fxp` project from the `Lesson18/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Directly below the event metadata and before the class definition, add two `SkinState` metadata tags defining states named `normal` and `disabled`.

```
[SkinState("normal")]  
[SkinState("disabled")]
```

You are specifying that anyone making a skin for your component must be able to handle these two states, or it is not to be considered a valid skin.

- 3 Inside the class definition, just below the variable declaration for the `_shoppingCart` property, add a public variable named `totalLabel` of type `Label`. Be sure to use code completion, but also be sure that you specify `spark.components.Label`.
- 4 Directly above the `totalLabel` property, add the `SkinPart` metadata, indicating that this particular part is required. Your code should look like this:

```
[SkinPart(required="true")]  
public var totalLabel:Label;
```

- 5 Add a new required `SkinPart` for `dataGroup` of type `DataGroup`.

```
[SkinPart(required="true")]  
public var dataGroup:DataGroup;
```


- 6 Add another new required SkinPart for quantityLabel of type Label.

```
[SkinPart(required="true")]  
public var quantityLabel:Label;
```

- 7 Finally, add an optional SkinPart for viewCartBtn of type Button.

```
[SkinPart(required="false")]  
public var viewCartBtn:Button;
```

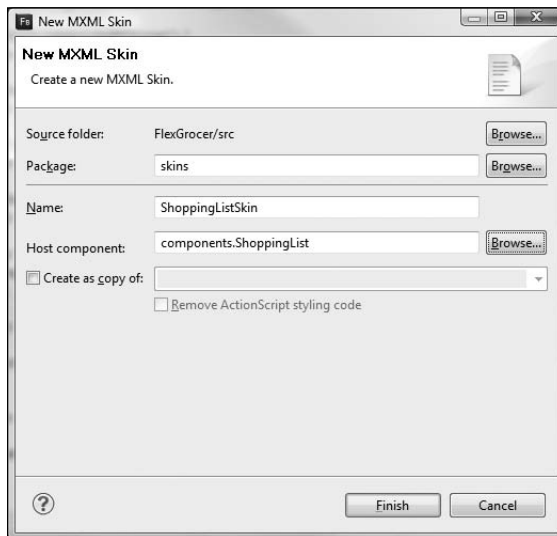
- 8 Save this class.

It should compile successfully without any errors or warnings.

Creating the Skin

You now have a component waiting to be skinned. It has the required skin parts and the skin states defined. In this section, you'll create a skin for the new component and apply it so that you can run the application and see some initial results.

- 1 Right-click the skins folder and choose New > MXML Skin from the pop-up menu.
- 2 Name the new skin **ShoppingListSkin**. Click Browse next to the Host component field and select your ShoppingList component.



- 3** Click Finish and a new skin is created for you.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <!-- host component -->
  <fx:Metadata>
    [HostComponent("components.ShoppingList")]
  </fx:Metadata>

  <!-- states -->
  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>

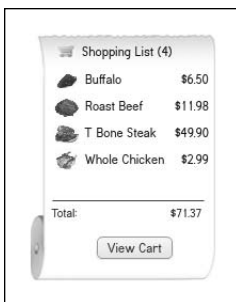
  <!-- SkinParts
  name=dataGroup, type=spark.components.DataGroup, required=true
  name=totalLabel, type=spark.components.Label, required=true
  name=quantityLabel, type=spark.components.Label, required=true
  name=viewCartBtn, type=spark.components.Button, required=false
  -->
</s:Skin>
```

Note that the HostComponent metadata was entered on your behalf, the required skin states were created based on the SkinState metadata in your ShoppingList class, and Flash wrote a comment in the code reminding you of the SkinParts you must have to be considered valid.

- 4** Just below the comment for the SkinParts, add an `<s:Image/>` tag with a source embedding the receipt.png @Embed('assets/receipt.png').

```
<s:Image source="@Embed('assets/receipt.png')"/>
```

This will load the background image for your new component. Here is a quick reminder of the skin you are about to build.



- 5 Below the Image, add an `<s:Label/>` tag with an `id` of `quantityLabel`. Set the `left` property to 50 and the `top` property to 10.

```
<s:Label id="quantityLabel" left="50" top="10"/>
```

Note that the `id` of `quantityLabel` is being used. This `id` is the same as the property marked with the `SkinPart` metadata in the `ShoppingList`. For every required `SkinPart` in the `ShoppingList`, you'll have a matching component here with that `id`.

- 6 Below the `quantityLabel`, add a tag pair for `<s:Scroller></s:Scroller>`. Set the `left` property to 22, the `top` property to 30, the `width` to 149, and the `height` to 115. You'll also set a property called `horizontalScrollPolicy` to `off`.

```
<s:Scroller left="22" top="30" width="149" height="115"
horizontalScrollPolicy="off">
</s:Scroller>
```

In Flex, not every object knows how to scroll its own content. Instead, you wrap these instances inside a `Scroller` to handle any scrolling needs. In this case, you are setting the size and position of the area you wish to scroll. By default, the `Scroller` scrolls horizontally and vertically. In this case, you only want vertical scrolling so `horizontalScrollPolicy` has been turned off.

- 7 Inside the `<s:Scroller></s:Scroller>` tag pair, add an `<s:DataGroup></s:DataGroup>` tag pair. Set the `id` of this `DataGroup` to `dataGroup`, one of your required skin parts. Set the `itemRenderer` property to `spark.skins.spark.DefaultItemRenderer`.

```
<s:Scroller left="22" top="30" width="149" height="115"
horizontalScrollPolicy="off">
  <s:DataGroup id="dataGroup"
    itemRenderer="spark.skins.spark.DefaultItemRenderer">
  </s:DataGroup>
</s:Scroller>
```

This `DataGroup` will be responsible for displaying the items in your `ShoppingCart`. For now, you are using the `DefaultItemRenderer`, which displays the text from your `toString()` method of your `ShoppingCartItem`. You'll customize this later.

- 8** Inside the `<s:DataGroup></s:DataGroup>` tag pair, set the layout property to an instance of the `VerticalLayout` class, setting the `gap` to `0`. Your code for the Scroller should look like this:

```
<s:Scroller left="22" top="30" width="149" height="115"
horizontalScrollPolicy="off">
  <s:DataGroup id="dataGroup"
    itemRenderer="spark.skins.spark.DefaultItemRenderer">
    <s:layout>
      <s:VerticalLayout gap="0"/>
    </s:layout>
  </s:DataGroup>
</s:Scroller>
```

- 9** Below the Scroller, draw a simple dividing line using MXML. Specify an `<s:Line>` tag pair with an `id` of `divider`. Set the `left` property to `22`, the `right` property to `10`, and the `top` to `155`. Inside the tag pair, set the `stroke` property to an instance of the `SolidColorStroke` with a color of `#545454` and a weight of `1`.

```
<s:Line id="divider" left="22" right="10" top="155">
  <s:stroke>
    <s:SolidColorStroke color="#545454" weight="1"/>
  </s:stroke>
</s:Line>
```

This code does nothing more than draw a dividing line before the total. You only have two labels and a button left until your skin is complete.

- 10** Add an `<s:Label/>` below the line with the text set to `Total:`, `left` set to `22`, `top` to `162`, color to `#0000FF`, and `fontSize` to `11`.

```
<s:Label text="Total:" left="22" top="162" color="#0000FF" fontSize="11"/>
```

- 11** Add another `<s:Label/>` with the `id` set to `totalLabel`, `right` set to `12`, `top` to `162`, color to `#0000FF`, and `fontSize` to `11`.

```
<s:Label id="totalLabel" right="12" top="162" color="#0000FF" fontSize="11"/>
```

This label will hold the actual formatted total on the shopping cart.

- 12** Finally, add an `<s:Button/>` with the `id` set to `viewCartBtn`, `label` set to `View Cart`, `horizontalCenter` to `12`, and `bottom` to `20`.

```
<s:Button id="viewCartBtn" label="View Cart" horizontalCenter="12" bottom="20"/>
```

This completes your skin for the moment. The code you added should look like the following snippet:

```
<s:Image source="@Embed('assets/receipt.png')"/>
<s:Label id="quantityLabel" left="50" top="10"/>

<s:Scroller left="22" top="30" width="149" height="
  115" horizontalScrollPolicy="off">
  <s:DataGroup id="dataGroup"
    itemRenderer="spark.skins.spark.DefaultItemRenderer">
    <s:layout>
      <s:VerticalLayout gap="0"/>
    </s:layout>
  </s:DataGroup>
</s:Scroller>

<s:Line id="divider" left="22" right="10" top="155">
  <s:stroke>
    <s:SolidColorStroke color="#545454" weight="1"/>
  </s:stroke>
</s:Line>

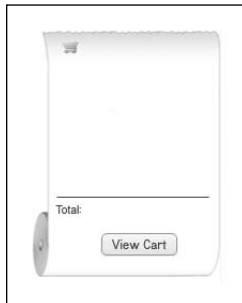
<s:Label text="Total:" left="22" top="162" color="#0000FF" fontSize="11"/>
<s:Label id="totalLabel" right="12" top="162" color="#0000FF" fontSize="11"/>
<s:Button id="viewCartBtn" label="View Cart" horizontalCenter="12" bottom="20"/>
```

13 Open `ShoppingView.mxml` and find the `ShoppingList` tag.

14 Add a property to this tag named `skinClass` and set it equal to `skins.ShoppingListSkin`.

```
<components:ShoppingList
  skinClass="skins.ShoppingListSkin"
  includeIn="State1"
  shoppingCart="{shoppingCart}"
  addProduct="addProductHandler(event)"
  viewCart="currentState='cartView'"/>
```

15 Save all your open files and ensure you do not have any errors or warnings. Run the `FlexGrocer` application and you should see the beginnings of your custom component displayed.



Adding Functionality to the Component

You created the stub for your new custom component and defined its visual appearance. Now it is time to add the final functionality so that both halves of the component work together. This is also the time when you'll need to understand just a bit about how Flash Player works internally as well as how to manage the internally asynchronous nature of components.

Handling Asynchronous for All

Flash Player is a single-threaded virtual machine. In the simplest sense, that means it does one thing at a time and regardless of how long it might take, it will never, ever interrupt code that is running. It always allows one task to finish before moving on to something else.

The problem with this philosophy is that if something takes a long time to do, it can cause Flash Player to stop updating the screen and mouse movements at a reasonable rate, creating a negative user experience.

To combat that issue, the Flex framework is event based and has an asynchronous component model. This means that certain aspects of what happens inside components happen at predetermined times when Flash Player is most optimally able to deal with changes. It also means that as a developer, you cannot make assumptions about when something is ready, complete, or otherwise available.

The Flex framework has prescribed ways to deal with this complexity. As a developer, if you embrace these concepts, things will go your way. If you try to do your own thing, the framework will find a way to punish you. Things may work seemingly well on your development machine but differently in production. Components may work in one circumstance but not another. Because all these issues have to do with timing that can change from machine to machine, it is imperative that you follow the rules.

- 1 Open the `ShoppingList.as` file that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer-PreFunction.fxp` project from the `Lesson18/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Just below the private `_shoppingCart` property, create a new private variable named `shoppingCartChanged` typed as a `Boolean`. Set it to a default value of `false`.

```
private var shoppingCartChanged:Boolean = false;
```

This is known as a *change flag* as its only purpose is to indicate the state of something. Internally this will be used to let your component know when it has a new `ShoppingCart` that must be displayed to the user.

- 3 Create two more private variables named `quantityChanged` and `totalChanged`, both typed as `Boolean` and with default values of `false`.

```
private var shoppingCartChanged:Boolean = false;
private var quantityChanged:Boolean = false;
private var totalChanged:Boolean = false;
```

These other change flags will be used for tracking when either the quantity or total need updating.

- 4 Inside the public setter for the `shoppingCart` property, immediately after `_shoppingCart` is set to value, set the `shoppingCartChanged` flag to `true`.

```
public function set shoppingCart(value:ShoppingCart):void {
    _shoppingCart = value;
    shoppingCartChanged = true;
}
```

- 5 Call the `invalidateProperties()` method that your class has due to inheritance.

```
public function set shoppingCart(value:ShoppingCart):void {
    _shoppingCart = value;
    shoppingCartChanged = true;
    invalidateProperties();
}
```

Everything that descends from `UIComponent` in Flex has this method available. This is one of several methods designed to help you deal with the asynchronous way Flex creates components. In Flex, skins can be added to and removed from components at runtime, so you cannot assume that all the parts of the skin are waiting and ready for your commands.

When you call `invalidateProperties()`, you are effectively asking the Flex framework to schedule a call to another special method named `commitProperties()` at a more opportune time. Flex manages the complexity of all the components that may want to do some work at any given time and calls them in the order most appropriate for performance.

- 6 Below the `shoppingCart` property setter, override a protected method named `commitProperties()`. This method takes no arguments. Immediately inside the method, call `super.commitProperties()`.

```
override protected function commitProperties():void {
    super.commitProperties();
}
```

This method is *eventually* called whenever you or any other code calls `invalidateProperties()`. Flex calls this method at an optimized time for your component to do the work it needs. In addition to the call you made to `invalidateProperties()`, other parts of the Flex framework also call this method. It will be called automatically whenever a new skin is added or removed.

- 7 Below the call to `super.commitProperties()`, write an `if` statement that checks if your `shoppingCartChanged` flag is true and if the `dataGroup` has already been created.

```
override protected function commitProperties():void {
    super.commitProperties();

    if ( shoppingCartChanged && dataGroup ) {
    }
}
```

The code inside this `if` statement will now only execute if your flag is true and if Flex has already created the `dataGroup`.

- 8 Inside the `if` statement, set the `shoppingCartChanged` flag to false. Then set the `dataProvider` of the `dataGroup` to `shoppingCart.items`.

```
override protected function commitProperties():void {
    super.commitProperties();

    if ( shoppingCartChanged && dataGroup ) {
        shoppingCartChanged = false;
        dataGroup.dataProvider = shoppingCart.items;
    }
}
```

All this code is mandatory. If you tried to access the `dataProvider` property of `dataGroup` before `dataGroup` existed, your application would crash. Memorize this pattern. Whenever a Flex component sets a property from outside the component (like your `shoppingCart` property) to another visual child component (like something in the skin), the `commitProperties()` method is used to ensure that the component will not crash due to timing issues.

- 9 Save your code and run the application. Items added to the cart via the Add and Remove buttons of the Products will appear in the cart list.

This is a great first step, but you have a lot more work to do.

- 10** Return to the `shoppingCart` setter. After the `shoppingCartChanged` flag is set to `true` but before `invalidateProperties()` is called, you need to write an `if` statement that checks if the shopping cart just passed to the function exists.

```
public function set shoppingCart(value:ShoppingCart):void {
    _shoppingCart = value;
    shoppingCartChanged = true;

    if ( _shoppingCart ) {
    }

    invalidateProperties();
}
```

It is always possible that a user working with your component passed in a `null` value. This check makes sure your code won't break when it tries to access the data. When developing components for reuse, you must code defensively.

- 11** Inside the `if` statement, add a new event listener to the `items` property of the `_shoppingCart`. You'll listen for a `CollectionEvent.COLLECTION_CHANGE` event and call a method name `handleItemChanged()` if this occurs.

```
public function set shoppingCart(value:ShoppingCart):void {
    _shoppingCart = value;
    shoppingCartChanged = true;

    if ( _shoppingCart ) {
        _shoppingCart.items.addEventListener(
            ↪ CollectionEvent.COLLECTION_CHANGE, handleItemChanged );
    }

    invalidateProperties();
}
```

This is the same code you wrote inside the `ShoppingCart` class so that the `ShoppingCart` would monitor changes in the `ShoppingCartItems`. This will serve a similar purpose here.

- 12** Create a new private method named `handleItemChanged()`. It will accept one parameter, an event of type `CollectionEvent`, and return nothing. Inside the method, set the `totalChanged` flag to `true` and the `quantityChanged` flag to `true`, and then call the `invalidateProperties()` method.

```
private function handleItemChanged( event:CollectionEvent ):void {
    totalChanged = true;
    quantityChanged = true;
    invalidateProperties();
}
```

This method will be called anytime you add, remove, or update any of the `ShoppingCartItem` instances. It sets these two changed flags to `true` and asks the Flex framework to call `commitProperties()` when it has the opportunity.

*** NOTE:** You never call `commitProperties()` yourself. You always call `invalidateProperties()` and let Flex decide when to call `commitProperties()`.

- 13** Create a new private variable named `currency` of type `spark.formatters.CurrencyFormatter` near the top of this class just between the `totalChanged` flag and the `totalLabel` `SkinPart` declaration.

```
private var currency:CurrencyFormatter;
```

This component is now going to take care of formatting the total before displaying it to the user.

*** NOTE:** You'll see there are three possible `CurrencyFormatters`. The one in the `flash.globalization` package is the low-level Flash Player version which provides much of the basic functionality. As mentioned in the `formatters` and `validators` package, there is also one in the `mx` and `Spark` package. Always defer to the `spark` package if you are not sure.

- 14** Find the `ShoppingList` class's constructor, and after the call to `super()` assign a new `CurrencyFormatter` class instance to the `currency` property. Then set the `useCurrencySymbol` property of the instance to `true`. Finally, call the `addStyleClient()` method and pass the `currency` instance.

```
public function ShoppingList() {
    super();
    currency = new CurrencyFormatter();
    currency.useCurrencySymbol = true;
    this.addStyleClient( currency );
}
```

Previously you created instances of the `CurrencyFormatter` through MXML. Here you are simply generating the `ActionScript` code that Flex would normally write on your behalf. The last line, `addStyleClient()` ensures that the `CurrencyFormatter` instance receives updates if you change the locale style in the application. This ensures the `CurrencyFormatter` here shows the correct currency type as the remainder of the application.

▶ TIP: If any of these lines show an error in Flash Builder, you likely imported the wrong version of `CurrencyFormatter`.

- 15** Return to the `commitProperties()` method. Below your other `if` statement, add a new `if` statement that checks if the `totalChanged` is `true` and if `totalLabel` exists. If it does, set the `totalChanged` flag to `false`.

```
if ( totalChanged && totalLabel ) {  
    totalChanged = false;  
}
```

- 16** Still inside the `if` statement but just below the code that sets `totalChanged` to `false`, set the `text` property of the `totalLabel` to the result of calling the `currency.format()` method, passing it the `shoppingCart.total` as an argument.

```
if ( totalChanged && totalLabel ) {  
    totalChanged = false;  
    totalLabel.text = currency.format( shoppingCart.total );  
}
```

Now each time the items in the `ShoppingCart` change, the shopping cart's total will be reformatted and the label in the skin will be updated.

- 17** Just after this `if` block, add one final `if` statement. Check if the `quantityChanged` flag is `true` and if the `quantityLabel` exists. If it does, set the `quantityChanged` flag to `false`.

```
if ( quantityChanged && quantityLabel ) {  
    quantityChanged = false;  
}
```

- 18** Still inside the `if` statement but just below the line of code that sets `quantityChanged` to `false`, set the `text` property of the `quantityLabel` to the result of concatenating the String `"Shopping List ("` + with the length of the shopping cart's items collection and a final `")"`.

```
if ( quantityChanged && quantityLabel ) {  
    quantityChanged = false;  
    quantityLabel.text = "Shopping List (" + shoppingCart.items.length + ")";  
}
```

Now each time the items in the `ShoppingCart` change, the shopping cart's quantity will be reformatted and the label in the skin will be updated.

- 19** Save and run the application. You can now add items to the shopping cart view using the `Product Add` and `Remove` buttons and see the `DataGroup`, `Quantity`, and `Total` update.

In the next section, you'll deal with drag and drop as well as the `View Cart` button.

Communicating with Events

Your component now updates and reflects data changes in the `ShoppingCart` instance. However, you still can't drag an item into the new `ShoppingList`, and you can't click the View Cart button. Those are your next tasks.

To perform them, you need to learn about another method available for override in `SkinnableComponent` descendants. That method is named `partAdded()`, and there is a corresponding method named `partRemoved()`. The `partAdded()` method will be called each time a new part of your skin is created and ready to access. The `partRemoved()` method is called when that skin part is removed and no longer part of the component.

- 1 Open the `ShoppingList.as` file that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer-PreDrag.fxp` project from the `Lesson18/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you skip a lesson or if you have a code issue you cannot resolve.

- 2 Just above the `commitProperties()` method, override the protected method named `partAdded`. This method accepts two parameters: the first is called `partName` of type `String` and the second is called `instance` of type `Object`. The method returns `void`. Immediately inside the method, call the `super.partAdded`, passing along the required arguments.

```
override protected function partAdded(partName:String, instance:Object):void {  
    super.partAdded( partName, instance );  
}
```

This method will be called each time a new skin part is built and ready for you to access. The `partName` will be the name of the `skinPart` (`dataGroup`, `totalLabel`, and so on). The `instance` will be a reference to the newly created object.

- 3 Just below the call to the super class, create an `if` statement that checks if the `instance` was `dataGroup`. Then create an `else` block that checks if it was `viewCartBtn`.

```
if ( instance === dataGroup ) {  
} else if (instance === viewCartBtn ) {  
}
```

You might have noticed the three equals sign in this code. This means strict equality. Normally in `ActionScript` the `==` looks for equality but allows the two items being compared to have different types so long as they evaluate to the same value. Strict equality indicates the value and type must match.

- 4** Inside the `if` statement for the `dataGroup`, add an event listener to the `dataGroup` instance for `DragEvent.DRAG_ENTER` and specify `handleDragEnter` as the listener. Add a second event listener to the `dataGroup` for `DragEvent.DRAG_DROP` and specify `handleDragDrop` as the listener for this event.

```
if (instance === dataGroup ) {
    dataGroup.addEventListener( DragEvent.DRAG_ENTER, handleDragEnter );
    dataGroup.addEventListener( DragEvent.DRAG_DROP, handleDragDrop );
} else if (instance === viewCartBtn ) {
}
```

This is just the ActionScript version of add event listeners to `dragEnter` and `dragDrop` in MXML.

- TIP:** When the `partAdded()` method is called by the Flex framework, it passes the `partName`, such as `dataGroup`, as well as an instance of type `Object`. Instead of adding your listener to `dataGroup` directly, you could have used `(instance as DataGroup).addEventListener()`. Those two statements would yield identical results.

- 5** Create a new private function named `handleDragEnter()` that accepts an event parameter of type `DragEvent` and returns `void`.

```
private function handleDragEnter( event:DragEvent ):void {
}
```

- 6** Inside this method call the `event.dragSource.hasFormat()` method and pass it the string `singleProduct`. If this method returns `true`, call `DragManager.acceptDragDrop()`, passing it the `event.target` typed as an `IUIComponent`.

```
private function handleDragEnter( event:DragEvent ):void {
    if(event.dragSource.hasFormat( "singleProduct" )){
        DragManager.acceptDragDrop( event.target as IUIComponent );
    }
}
```

This method should look familiar. This is nearly the same method you wrote for the `dragEnter` handler previously in `ShoppingView`. Now you are just handling everything in ActionScript.

- 7** Create a new private function named `handleDragDrop()` that accepts an event parameter of type `DragEvent` and returns `void`.

```
private function handleDragDrop( event:DragEvent ):void {
}
```

- 8** Inside this method create a new local variable named `product` of type `Product`, and assign its initial value to the result of the `event.dragSource.dataForFormat()` method, passing it the string `singleProduct`. Cast the result as a `Product` object.

```
private function handleDragDrop( event:DragEvent ):void {
    var product:Product =
        ➤ event.dragSource.dataForFormat( "singleProduct" ) as Product;
}
```

This method should also look familiar. It is again nearly the same method you wrote for the `dragDrop` handler earlier in `ShoppingView`.

- 9** Just after getting the `Product` instance from the drag event, create a new local variable named `prodEvent` of type `ProductEvent`. Assign its value to a new instance of the `ProductEvent`, passing the string `addProduct` to the first parameter and the `Product` object to the second.

```
var prodEvent:ProductEvent = new ProductEvent( "addProduct", product );
```

In the very beginning of this exercise, you told the Flex compiler you would dispatch a product event. You are about to fulfill that promise.

- 10** As the last line of this method, dispatch the `prodEvent` event.

```
private function handleDragDrop( event:DragEvent ):void {
    var product:Product =
        ➤ event.dragSource.dataForFormat( "cartFormat" ) as Product;
    var prodEvent:ProductEvent = new ProductEvent( "addProduct", product );
    dispatchEvent( prodEvent );
}
```

On a successful drag-and-drop operation, your code now dispatches an event indicating that the product should be added to the cart.

- 11** Return to the `partAdded()` method. In the `else` clause for the `viewCartBtn` part, add an event listener to the `viewCartBtn` instance for the `MouseEvent.CLICK` event, passing `handleViewCartClick` as the listener. Here is the final `partAdded()` method:

```
override protected function partAdded(partName:String, instance:Object):void {
    super.partAdded( partName, instance );

    if ( instance === dataGroup ) {
        dataGroup.addEventListener( DragEvent.DRAG_ENTER, handleDragEnter );
        dataGroup.addEventListener( DragEvent.DRAG_DROP, handleDragDrop );
    } else if ( instance === viewCartBtn ) {
        viewCartBtn.addEventListener( MouseEvent.CLICK, handleViewCartClick );
    }
}
```

- 12** Create a new private function named `handleViewCartClick()` that accepts an event parameter of type `MouseEvent` and returns `void`.

```
private function handleViewCartClick( event:MouseEvent ):void {
}
```

- 13** Inside this method create a new local variable named `viewEvent` of type `Event`. Assign it to a new instance of the `Event` class, passing the string `viewCart`. Finally, dispatch the event.

```
private function handleViewCartClick( event:MouseEvent ):void {
    var viewEvent:Event = new Event( "viewCart" );
    dispatchEvent( viewEvent );
}
```

This will dispatch the `viewCart` event that you defined long ago at the beginning of this component.

- 14** Save and test your application. You should now be able to add items to the shopping list by dragging them, and the View Cart button should switch to the datagrid version of the view.

Cleaning Up After Yourself

Your component now works quite well, but there is a problem. Skins in Flex can be changed at runtime. You are adding event listeners to a number of components in the skin but not cleaning up after yourself.

The same is true of the data passed to the `shoppingCart`. Right now you add an event listener; however, if someone provided a new `ShoppingCart` instance, you would be listening to two collections for changes instead of just the most recent.

- 1 Open the `ShoppingList.as` file that you used in the previous exercise.
- 2 Copy the `partAdded()` method in its entirety. Paste it just below the existing method. Change the name of the function to `partRemoved` and change the call to the super class to `partRemoved` as well.

```
override protected function partRemoved(partName:String, instance:Object):void
{
    super.partRemoved( partName, instance );

    if ( instance === dataGroup ) {
        dataGroup.addEventListener( DragEvent.DRAG_ENTER, handleDragEnter );
        dataGroup.addEventListener( DragEvent.DRAG_DROP, handleDragDrop );
    } else if ( instance === viewCartBtn ) {
        viewCartBtn.addEventListener( MouseEvent.CLICK, handleViewCartClick );
    }
}
```

You should have just changed `partAdded` to `partRemoved` in *two* places. If you changed it a different number of times, recheck before proceeding.

- 3** Inside the `partRemoved()` method, change all the calls to `addEventListener()` to `removeEventListener()`. Keep the parameters the same.

```
override protected function partRemoved(partName:String, instance:Object):void
{
    super.partRemoved( partName, instance );

    if ( instance === dataGroup ) {
        dataGroup.removeEventListener( DragEvent.DRAG_ENTER, handleDragEnter );
        dataGroup.removeEventListener( DragEvent.DRAG_DROP, handleDragDrop );
    } else if ( instance === viewCartBtn ) {
        viewCartBtn.removeEventListener( MouseEvent.CLICK, handleViewCartClick );
    }
}
```

You should have just changed `addEventListener` to `removeEventListener` in *three* places. If you changed it a different number of times, recheck before proceeding. Now each time a part is removed, it removes the accompanying event listeners.

- 4** Find the `shoppingCart` setter function.

Currently this function adds an event listener each time it is called. You'll now also remove the old event listener.

- 5** Copy the `if` block that checks if the `_shoppingCart` property exists and adds an event listener. Paste it as the first line of this method.

```
public function set shoppingCart(value:ShoppingCart):void {
    if ( !_shoppingCart ) {
        _shoppingCart.items.addEventListener(CollectionEvent.COLLECTION_CHANGE,
            ↪ handleItemChanged );
    }

    _shoppingCart = value;
    shoppingCartChanged = true;

    if ( _shoppingCart ) {
        _shoppingCart.items.addEventListener(CollectionEvent.COLLECTION_CHANGE,
            ↪ handleItemChanged );
    }

    invalidateProperties();
}
```

This method now adds two event listeners, which is worse than before.

- 6 Change the first call to `_shoppingCart.items.addEventListener()` to `removeEventListener()` instead.

```
public function set shoppingCart(value:ShoppingCart):void {
    if ( !_shoppingCart ) {
        _shoppingCart.items.removeEventListener(CollectionEvent.COLLECTION_CHANGE,
            ▶ handleItemChanged );
    }

    _shoppingCart = value;
    shoppingCartChanged = true;

    if ( !_shoppingCart ) {
        _shoppingCart.items.addEventListener(CollectionEvent.COLLECTION_CHANGE,
            ▶ handleItemChanged );
    }

    invalidateProperties();
}
```

This code now checks to see if there was already a `shoppingCart` with an event listener. If so, it removes it before adding a listener to a new one.

- 7 Save and run your application. Make sure it performs as it did before.

The `ShoppingList` is finished. All that is left is to customize the way the `DataGroup` instance in the skin displays data.

Creating a Renderer for the Skin

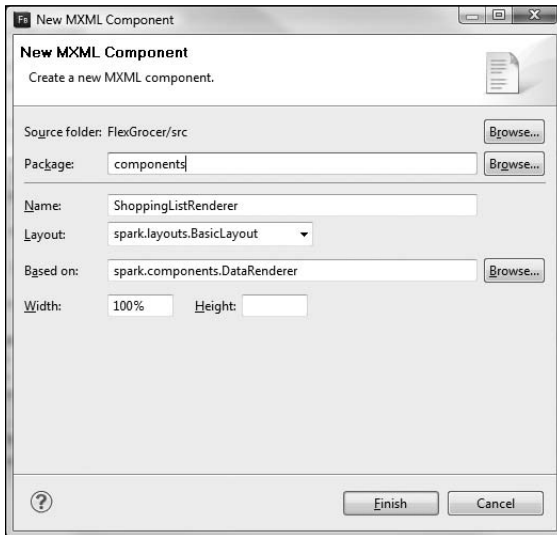
The last step to finish up the presentation of this component is to create a custom renderer and apply it to the `DataGroup` that the `ShoppingListSkin` will use to render its data. This will complete the desired look of the component.

As you may remember from Lesson 10, “Using DataGroups and Lists,” extending `DataRenderer` is a fast and easy way to create a custom renderer for a `DataGroup`.

- 1 Open the `FlexGrocer` project that you used in the previous exercise.

Alternatively, if you didn't complete the previous lesson or your code is not functioning properly, you can import the `FlexGrocer-PreRenderer.fxp` project from the `Lesson18/intermediate` folder. Please refer to the appendix for complete instructions on importing a project should you ever skip a lesson or if you ever have a code issue you cannot resolve.

- 2 In the Package Explorer, right-click the components package and choose New MXML Component. Name the component **ShoppingListRenderer**, choose BasicLayout for the Layout, and specify DataRenderer for the Based on value. Set the Width to **100%** and remove the value for the Height.



- 3 Beneath the declarations tag, create an `<fx:Script>` tag pair. Inside the Script block, create a new bindable private variable named `item` of type `ShoppingCartItem`.

```
<fx:Script>
  <![CDATA[
    import cart.ShoppingCartItem;
    [Bindable]
    private var item:ShoppingCartItem;
  ]]>
</fx:Script>
```

- 4 Still inside the Script block, override the public setter for `data`. Set the `item` property you just created to the value typed as a `ShoppingCartItem`.

```
override public function set data(value:Object):void{
  this.item = value as ShoppingCartItem;
}
```

- 5 Inside the declarations block, create a new `<s:CurrencyFormatter/>` tag with an `id` of `currency` and set the `useCurrencySymbol` property to `true`.

```
<s:CurrencyFormatter id="currency" useCurrencySymbol="true"/>
```

- 6** Below the Script block in MXML, create a new `<s:Image/>` tag. Set its source equal to `assets/{item.product.imageName}`. Then set its width to 25 and height to 25.

```
<s:Image source="assets/{item.product.imageName}" width="25" height="25"/>
```

- 7** Below the image, create a new `<s:Label/>` tag. Set its left to 30, top to 5, right to 30, and text to `{item.product.prodName}`.

```
<s:Label left="30" top="5" right="30" text="{item.product.prodName}"/>
```

- 8** Below the label, create another new `<s:Label/>` tag. Set its right to 1, top to 5, and text to `{currency.format(item.subtotal)}`.

```
<s:Label right="1" top="5" text="{currency.format(item.subtotal)}"/>
```

- 9** Open the `ShoppingListSkin.mxml` file from the skins package.

- 10** Find the `itemRenderer` of the `DataGroup` tag, which is currently set to `spark.skins.spark.DefaultItemRenderer`, and set it instead to `components.ShoppingListRenderer`.

- 11** Save and run your application.

You should now have a completely styled custom component.

What You Have Learned

In this lesson, you have:

- Learned the concepts of custom ActionScript components (page 420)
- Performed an extensive refactor (pages 424–429)
- Created an ActionScript skinnable component (pages 424–429, 439–450)
- Created your own skin (pages 434–438)
- Used the Scroller class (page 436–438)
- Learned to manage skin parts and life cycle (pages 445–452)

This page intentionally left blank

This appendix contains the requirements and instructions for you to complete the exercises in this book. It covers the following:

- Minimum system requirements
- Software installation
- Importing projects

Minimum System Requirements

General Requirements

You must have a recent version of one of the following browsers installed:

- Internet Explorer
- Mozilla Firefox
- Safari
- Opera
- Google Chrome

Windows

- 2 GHz or faster processor
- 1 GB of RAM (2 GB recommended)
- Microsoft Windows XP with Service Pack 3, Windows Vista Ultimate or Enterprise (32 or 64 bit running in 32-bit mode), Windows Server 2008 (32 bit), or Windows 7 (32 or 64 bit running in 32-bit mode)
- 1 GB of available hard-disk space
- Java Virtual Machine (32 bit): IBM JRE 1.5, Sun JRE 1.5, IBM JRE 1.6, or Sun JRE 1.6
- 1024x768 display (1280x800 recommended) with 16-bit video card
- Flash Player 10.2 or later

Macintosh

- Intel processor based Mac
- OS X 10.5.6 (Leopard) or 10.6 (Snow Leopard)
- 1 GB of RAM (2 GB recommended)
- 1 GB of available hard-disk space
- Java Virtual Machine (32 bit): JRE 1.5 or 1.6
- 1024x768 display (1280x800 recommended) with 16-bit video card
- Flash Player 10.2 or later

▶ **TIP:** To check your Flash Player version, go to www.adobe.com, right-click the main ad banner, and select About Flash Player; or visit www.adobe.com/software/flash/about.

APPENDIX

Setup Instructions

Be sure to complete the installation of all required files before working through the lessons in the book.

Software Installation

There are three phases of the installation:

- Installing Flash Builder
- Installing lesson files
- Installing Flash Debug Player (if a problem exists)

Installing Flash Builder

If you do not have Flash Builder 4.5 installed, step through the following installation directions:

- 1 Browse to http://www.adobe.com/go/try_flashbuilder and choose your platform.
- 2 Click Download now.
- 3 Install Flash Builder, accepting all the default options. The trial period on Flash Builder 4.5 is 60 days.

*** NOTE:** If you happen to be:

- A student, faculty, or staff of eligible education institution
- A software developer who is currently unemployed then you are eligible for a free Flash Builder license. Visit <https://freeritools.adobe.com/>.

Installing Lesson Files

Once again, it's important that all the required files are in place before working through the lessons in the book.

From the CD included with your book, copy the `flex4tfs` directory to the root of your drive.

In this directory, there is a subdirectory named `FlexGrocer`, in which you will be doing most of your work. Also included are directories for each lesson in the book with starting code and completed code for the work you do in the lesson. Some lessons may also have an intermediate directory, which highlights major steps in the lesson, or an independent directory, which holds projects that are unrelated to the main application but demonstrates an important technique or concept.

Installing Flash Debug Player

At various times in the book, you'll be using features of Flash Debug Player. If you receive a notice saying you do not have Flash Debug Player installed, follow these steps to install it:

Windows

- 1 Locate the Flash Player (Player) install directory:

applicationInstallDirectory/Adobe/Adobe Flash Builder 4.5/player/win.

*** NOTE:** In a default 32-bit installation, this directory in Windows is C:\Program Files\Adobe\Adobe Flash Builder 4.5\player\win.

- 2 To install Flash Debug Player for Internet Explorer, run the Install Flash Player 10.2 ActiveX.exe program. For other versions of web browsers, run the Install Flash Player 10.2 Plugin.exe program.

Macintosh

- 1 Locate the Flash Player (Player) install directory:

applicationInstallDirectory/Adobe/Adobe Flash Builder 4.5/player/mac/10.2.

*** NOTE:** In a default installation, this directory would be /Applications/Adobe/Adobe Flash Builder 4/player/mac/10.2.

- 2 To install Flash Debug Player, run Flash Player Plugin Debugger.dmg.

▶ TIP: In rare instances, you might run the appropriate installer and still get the message that you don't have the debug version of the player. In this case, uninstall the version you currently have by using the information you'll find at

http://kb.adobe.com/selfservice/viewContent.do?externalId=tn_14157

and then follow the steps above to reinstall.

Importing Projects

Anytime during the course of this book that your code is no longer working, or you simply wish to jump ahead or revisit a lesson, you can import the appropriate FXP (Flex Project) for that lesson.

The FXP format is an archive file that includes project information, files, and folders. The project includes all assets and dependencies for your Flex project.

Each lesson of the book has start and complete files representing the code for that lesson at its beginning and end if all steps are executed. The FXP files for the lessons have this format:

```
driveRoot/flex4TFS/LessonXX/start/FlexGrocer.fxp
driveRoot/flex4TFS/LessonXX/complete/FlexGrocer.fxp
```

Some lessons have intermediate files that represent major steps internal to the lesson. Those follow this format:

```
driveRoot/flex4TFS/LessonXX/intermediate/FlexGrocer-NameOfStep.fxp
```

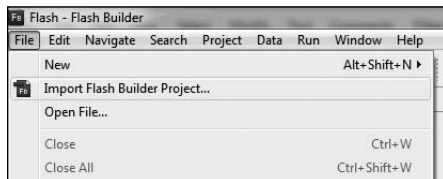
Finally, a few lessons have independent files that represent small applications intended to demonstrate a singular concept but that are not directly related to the code in your FlexGrocer project. Those follow this format:

```
driveRoot/flex4TFS/LessonXX/independent/projectName.fxp
```

*** NOTE:** driveRoot represents your root drive—for instance, C:\ on Windows.

Importing Lesson Files

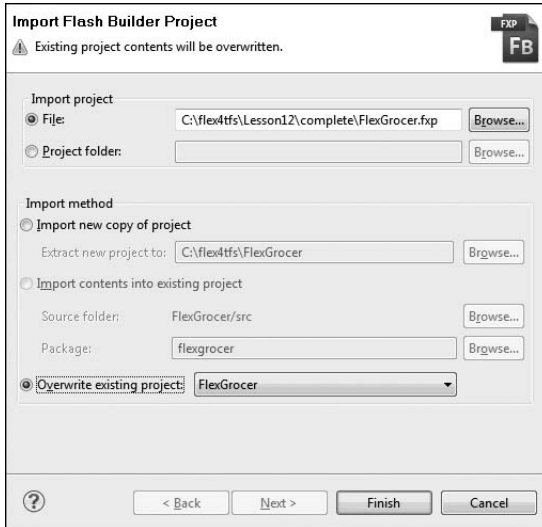
You import an FXP file by choosing File > Import Flash Builder Project.



When you attempt to import a project in Flash Builder, the IDE determines whether you already have a project with the same Universally Unique Identifier (UUID). This will occur if you have previously imported any lesson.

When You Have an Existing Project

In this case, Flash Builder will allow you to overwrite the existing project with a fresh copy. You simply need to choose the Overwrite existing project radio button and choose the FlexGrocer project from the drop-down menu. Then click Finish.



This is the ideal way to import a new lesson as it simply overwrites the older code and leaves you with a single FlexGrocer project.

When You Don't Have an Existing Project

If you have never imported the FlexGrocer project before, the overwrite option will not be available.

In this case, choose the Import new copy of project radio button and set the “Extract new projects to input” field to *driveRoot/flex4tfs/FlexGrocer*. Then click Finish.

Import Flash Builder Project
Import a Flash Builder project from an FXP, ZIP, or project folder.

Import project

File: C:\flex4tfs\Lesson12\complete\FlexGrocer.fxp **Browse...**

Project folder: **Browse...**

Import method

Import new copy of project

Extract new project to: C:\flex4tfs\FlexGrocer **Browse...**

Import contents into existing project

Source folder: **Browse...**

Package: **Browse...**

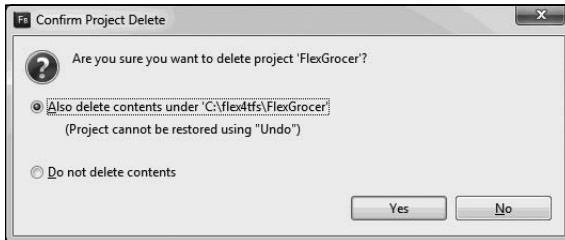
Overwrite existing project:

? < Back Next > Finish Cancel

When Flash Builder Prevents Overwriting an Existing Project

If you already have a project named FlexGrocer in your workspace, and Flash Builder does not recognize this as the same project identifier, it will not allow you to overwrite that project. In this case, simply right-click the FlexGrocer project and choose Delete from the Package Explorer.

A dialog box will appear and ask if you want to delete the Flash Builder project only or also the files for the project from the file system. Choose the Also delete contents radio button and click Yes.



After deleting these files, repeat these directions to import the needed project.

Index

- @ (attribute) operator, 131
 - { } (braces), 81, 89, 98, 176–177
 - :
 - colon, 23
 - = (equal sign), 23
 - > (greater-than sign), 30
 - ? (question mark), 146
 - " (quotation marks), 68, 97
 - / (slash), 30, 31
 - _ (underscore), 177
 - " ; (escape code), 68
 - .. (descendant) operator, 132
 - .
 - (dot) operator, 130
- A**
- AATC (Adobe Authorized Training Center), xxii
 - absolute positioning, 58, 59
 - ACA (Adobe Certified Associate), xxii
 - acceptDragDrop() method, 322
 - ACE (Adobe Certified Expert), xxii
 - ACI (Adobe Certified Instructor), xxii
 - action item controls, 303
 - ActionScript
 - Array instances, 184–185
 - classes, 142, 143, 427–429
 - components (See ActionScript components)
 - dispatching events in, 178–179
 - Drawing API, 404
 - event handling with, 98–99
 - and Flash Platform, 12, 13
 - for loops, 162
 - power of, 12
 - purpose of, 13
 - triggering validation from, 363
 - XML support, 127
 - ActionScript components, 419–452
 - adding functionality to, 439–450
 - choosing base class for, 426
 - complexity of building, 420
 - creating visuals for, 432–438
 - defining, 424–431
 - overview, 420–422
 - reasons for making, 422–424
 - specifying skin requirements for, 432–434
 - types of, 421
 - ActionScript Virtual Machine (AVM), 13
 - addData() method, 322
 - addEventListener() method, 275
 - addInts() method, 40
 - addItem() method, 157, 158, 160, 166–167
 - Add To Cart button, 158, 161, 167
 - addTextArea event handler, 108
 - Adobe
 - certification levels, xxii
 - Certified Associate exams, xxi
 - Community Help, xx–xxii
 - Creative Suite, 14
 - Developer Connection, xxi
 - Flash (See Flash)
 - Flex (See Flex)
 - Labs, xxii
 - Marketplace & Exchange, xxii
 - open source site, 81
 - TV, xxi
 - Adobe Advanced Training series, xviii
 - Adobe Flex 4.5 MXML and ActionScript Language Reference (ASDoc), 41, 79, 208, 378
 - Adobe Integrated Runtime (AIR), 12
 - Adobe Training from the Source series, xviii
 - AIR (Adobe Integrated Runtime), 12
 - AJAX (Asynchronous JavaScript and XML), 8, 9–10
 - anchors, layout, 59
 - animation tools, 12
 - API (application programming interface), 79
 - Application container, 46
 - application files. See also applications
 - basic elements of, 22
 - comparing versions of, 31–32
 - creating, 21–22
 - naming, 21
 - organizing, 18–24
 - saving, 31
 - viewing, 18, 21–22
 - viewing errors in, 32–33
 - application programming interface (API), 79
 - applications. See also application files
 - building detail view of, 85–87
 - changing internal structure of, 73–75
 - controlling view states for, 68–70
 - customizing with skins, 399–417 (See also skins)
 - customizing with styles, 375–397 (See also styles)
 - debugging, 34–41
 - desktop, 4–5, 311
 - displaying images in, 81–84
 - displaying/managing data for, 144–150, 292
 - dividing into modules, 207
 - embedding fonts in, 388, 390
 - enterprise server, 10
 - evolution of, 3, 4–6
 - improving architecture of, 213–221
 - laying out interface for, 50–58
 - mainframe, 4
 - maintainability of, 5, 8

- minimum height/width for, 24
 - refactoring, 71–75, 101–103
 - rich Internet (*See* RIAs)
 - running, 28–33
 - saving, 187
 - tightly coupled, 262
 - viewing hierarchy of, 57
 - web (*See* web applications)
 - Web 2.0, xii
 - working with view states for, 63–70
 - Application tag, 22–23, 31
 - architecture
 - client-side, 95
 - loosely coupled, 262–263
 - model-view-controller (MVC), 88, 212–213
 - page-based, 4, 6–7, 8
 - service-oriented, 5
 - arguments, 147
 - ArrayCollection, 184–203
 - and cursors, 198
 - and data binding, 184–185
 - filtering items in, 202–203
 - populating, 185–192
 - sorting items in, 194–198
 - using data from, 192–193
 - vs. ArrayList, 246
 - ArrayList, 185, 246
 - Array notation, 192
 - arrays. *See also* ArrayCollection
 - and data binding, 183–184
 - determining number of items in, 202–203
 - for shopping cart items, 159–167
 - sorting items in, 194–198
 - using items in, 192–193
 - AS3 Drawing API, 404
 - ASDoc (*Adobe Flex 4.5 MXML and ActionScript Language Reference*), 41, 79, 208, 378
 - aspect ratio, 83
 - asynchronous component model, 439
 - Asynchronous JavaScript and XML (AJAX), 8, 9–10
 - attribute (@) operator, 131
 - attributes, 23, 30, 98
 - Auto-Generation, 181–183
 - AVM (ActionScript Virtual Machine), 13
- B**
- base class, 426
 - BasicLayout object, 47, 58, 90
 - [Bindable] metadata tag, 135–136, 145, 149, 157, 173, 176–177
 - bindings, 98. *See also* data binding
 - BitmapFill class, 408
 - boldface text/code, xvi
 - Boolean values, 151–152, 163
 - BorderContainer, 46
 - braces ({}), 81, 89, 98, 176–177
 - Breakpoint Properties view, 40
 - breakpoints
 - enabling conditional, 40
 - removing, 39, 107
 - setting, 34, 35, 104, 187
 - turning on/off, 39
 - Breakpoints view, 36
 - browsers, web, 4, 5
 - bubbling, event, 274–279
 - Build Automatically option, 28
 - buildProduct() method, 151, 152, 199
 - business applications
 - evolution of, 4–6
 - maintainability of, 5, 8
 - role of computers in, 3
 - button_clickHandler() method, 35, 37
 - Button control, 57
 - buttons
 - creating skin for, 400–404
 - customizing for different states, 407–413
- C**
- calculateSubtotal() method, 156, 204
 - calculateTotal() method, 164–165
 - camel case syntax, 382
 - capture phase, event flow, 275
 - CartGrid component, 295–296
 - Cascading Style Sheets. *See* CSS
 - case-sensitivity, 21, 50, 68
 - categories, filtering products by, 257–259
 - categories property, 231
 - CDATA (character data) blocks, 99, 102, 110
 - CD-ROM, xvii, xviii
 - centralized data processing, 4
 - change events, 258, 287–288, 442
 - change flags, 440
 - character data (CDATA) blocks, 99, 102, 110
 - Checkout application, 89–92
 - Checkout button, 56, 59, 345
 - checkout form, 89–92
 - checkout process, 335–358
 - adding billing info page to, 345–350
 - creating OrderEvent object for, 355–358
 - creating OrderInfo object for, 335–336
 - creating review page for, 350–355
 - creating user views for, 337–345
 - overview, 335
 - validating postal codes during, 369–372
 - CheckoutView component, 337–345
 - children, 46, 57, 128
 - child tags, 49
 - classes, 141–167. *See also* specific classes
 - and ActionScript, 142, 143
 - basics of building, 143
 - as blueprint for objects, 141
 - categories of, 274

- classes (*continued*)
 - constructors of, 143
 - creating, 141, 154, 427–429
 - custom, 429–431
 - defining, 49–50, 143
 - naming, 143
 - properties of, 143, 147
 - reference document for, 41
 - vs. properties, 49–50
- class hierarchy, 208–209
- class instances, 49
- class keyword, 145
- class selectors, 382, 383, 384
- clickHandler() function, 99, 100
- client/server applications, 3, 4–5, 7
- client-side architecture, 95
- clone() method, 271
- code blocks, xvii
- code completion, 109, 110, 150
- code hinting, 29, 30, 98
- code line numbers, 27
- ColdFusion, xiv
- CollectionEvent class, 287–288
- collections
 - examples of, 194
 - filtering items in, 202–203
 - sorting items in, 194–198
 - using cursors in, 198–199
- colon (:), 23
- color
 - background, 376, 382, 393, 394, 404
 - highlight, 375, 402
 - label, 391, 392, 393
 - logo, 54
 - rollover, 379–381, 382, 383
 - text, 378
- colorName label, 171
- columns
 - in DataGrid, 292, 294, 297, 299
 - in layout objects, 46
- commitProperties() method, 440, 441, 443
- Community Help application, xx–xxii
- compiler, 176–183
- compiler arguments, 142
- compile-time errors, 33, 150
- components, 207–238. *See also* specific components
 - ActionScript (*See* ActionScript components)
 - advantages of, 212
 - applying styles to, 379–381
 - broadcasting events from, 263
 - changing look of, 400–404, 432–438
 - complexity of building, 420
 - creating, 105, 209–212, 230–238
 - creating directory of reusable, 221–230
 - declaring events for, 267–269
 - defining, 46, 424–431
 - drag-and-drop, 313
 - facilitating use of, 212
 - generic, 425
 - hierarchy of, 208–209
 - list-based, 321
 - to manage loading data, 230–238
 - and MVC architecture, 212–213
 - MXML, 208–213, 420
 - non-visual, 230–238
 - positioning, 46, 55
 - purpose of, 207
 - specifying skin requirements for, 432–434
 - types of, 421
 - visual, 213–230, 274
- Components view, 54
- composed containers, 72–73
- computer applications. *See* applications
- conditional breakpoints, 40
- configuration files, 23
- constraint-based layouts, 55, 58–63
- ConstraintLayout object, 47
- constructors, 143, 147, 148, 155–156
- containers
 - combining layout objects and, 48
 - composed, 72–73
 - finding, 65
 - positioning elements in, 58–59
 - purpose of, 46
 - size considerations, 62
 - types of, 46–47
- control bars, 51, 53, 57
- controllers, 212
- controls
 - accessing properties for, 81
 - APIs for, 79
 - assigning IDs to, 81
 - positioning, 64
 - simple (*See* simple controls)
- Cookbooks, xxi
- cookies, 7
- copyright label, 60
- createCursor() method, 199
- creationComplete event, 107–111
- Creative Commons License, 388
- CreditCardInfo component, 345–350
- cross-domain policy files, 122
- CSS (Cascading Style Sheets)
 - how namespaces are defined by, 389
 - inheritance, 381
 - standards, 384
 - styling with, 377–378, 382
- CSS files
 - creating SWF files from, 395–396
 - setting styles via, 386–394
 - sharing between applications, 386
- curly brackets ({}). *See* braces
- CurrencyFormatter class, 362–363, 364–368, 443
- currentState property, 68
- cursors
 - defined, 198
 - general steps for using, 198–199
 - refactoring to search with, 198–201
 - removing items with, 201–202
- custom classes, 429–431

- custom components. *See also* components
 - advantages of, 212
 - example, 210–211
 - facilitating use of, 212
 - and MVC architecture, 212–213
 - steps for creating, 209–210
 - ways of using, 207, 282
- custom event classes, 269–270
- D**
- data. *See also* datasets
 - allowing users to interact with, 292
 - in `ArrayCollections`, 192–193
 - creating components to manage loading, 230–238
 - debugging, 149
 - from event objects, 100–103
 - externalizing, 114–116
 - filtering, 202–203
 - manipulating shopping cart, 159–167
 - modifying, 109–111
 - nodes vs. attributes, 186
 - passing, when calling event handler, 99–100
 - passing, with events, 269–270
 - retrieving, 108, 120, 126
 - security issues, 122–123
- data binding, 169–205
 - and arrays, 183–184
 - binding simple controls using, 81
 - breaking complex, 173–176
 - breaking simple, 170–173
 - curly bracket syntax for, 81, 89 (*See also* braces)
 - as events, 179–181
 - and Flex formatters, 362, 363
 - implications of, 183–184
 - importance of, 170
 - linking data structure to simple control using, 88–89
 - populating text controls using, 81
 - proxying strategy, 184–185
 - purpose of, 88
 - two-way, 334
- `dataChanged` event, 249
- `dataForFormat()` method, 322
- `DataGrid`, 291–309
 - adding, 65–67
 - considerations, 292
 - default behavior of, 299
 - displaying shopping cart with, 292–308
 - dragging/dropping between List and, 315–321
 - purpose of, 292
- `DataGroup`, 245–255
 - creating custom renderer for, 450–452
 - creating `ProductList` from, 282–285
 - enabling virtualization for, 254–255
 - implementing `itemRenderer`, 246–251
 - purpose of, 242
 - simple example, 245–246
 - using in `ShoppingView`, 251–253
 - vs. `Group` class, 245
- data models, 88
- data nodes, 186
- data persistence, 15
- `dataProvider` property, 242
- `DataRenderer` class, 251, 256, 450
- datasets
 - defined, 241
 - populating List control with, 242–245
 - using `DataGroup` with, 245–246
 - virtualization of, 254
- data structures, 81
- data transfer objects (DTOs), 143
- debugger, 34–41, 104
- debugging. *See also* breakpoints; errors
 - data binding, 171–172, 175
 - data structures, 149
 - example, 34–41
 - and Local History feature, 31–32
 - rolling back to previous versions, 31–32
- Debug perspective, 105
- Debug view, 35, 38–39
- Declarations tag, 29
- `DefaultItemRenderer` class, 246
- default state, 63
- descendant (`.`) operator, 132
- descendant selectors, 382, 390–391
- Design button, 22
- Design view, 25, 53–56
- desktop applications, 4–5, 311
- detail view, 85–87
- DHTML (Dynamic HTML), 5, 9
- `dispatchEvent()` method, 263, 270
- dispatching events, 178–179, 183, 263–267
- display list, 274
- `DisplayObjects`, 274
- Document Object Model (DOM), 5
- `doDrag()` method, 322
- DOM (Document Object Model), 5
- dot-com boom, 7
- dot operator, 130
- double quotes (`"`), 97
- Drag and Drop Manager, 311, 312–313
- `dragComplete` event, 315
- `dragDrop` event, 316
- drag/drop operations, 311–331
 - between `DataGrid` and List, 315–321
 - and HTML, 4–5
 - phases, 312
 - in shopping cart, 326–331
 - terminology, 312
 - between two lists, 313–315
 - using non-drag-enabled component in, 321–326
- `dragEnabled` property, 313, 314, 315
- `dragEnter` event, 316
- `DragEvents`, 101

dragExit event, 316
 drag initiator, 312, 315
 DragManager class methods, 322
 dragOver event, 316
 drag proxy, 312
 drag source, 312
 DragSource class, 312, 314, 315, 322
 Drawing API, AS3, 404
 drawRect() method, 406
 dropEnabled property, 313, 315
 drop target, 312, 316
 DTOs (data transfer objects), 143
 dumb terminals, 4
 Dynamic HTML (DHTML), 5, 9
 dynamic interfaces, 71
 dynamic XML data, 133–137

E

E4X (ECMAScript for XML), 127–133
 Eclipse platform, 14, 17, 36
 Eclipse project, 10, 14
 ECMAScript for XML (E4X), 127–133
 ECMA standard, 127
 e-commerce application. *See also* FlexGrocer application
 laying out, 50–58
 using drag and drop in, 311–331
 working with view states in, 63–70
 editors
 defined, 24
 example, 22
 expanding, 25
 inline, 296–299
 opening/closing, 24
 showing code line numbers in, 27
 viewing errors in, 32–33
 element selectors, 382
 embedded fonts, 388, 390
 embedded XML, 114–119
 @Embed directive, 84
 end users, 8

enterprise server applications, 10
 equal sign (=), 23
 error messages, 89
 errors. *See also* debugging
 and Build Automatically option, 28
 how Flash Builder reports, 27
 viewing, 32–33
 escape code (" ;), 68
 event-based programming model, 95, 96–97
 event bubbling, 274–279
 Event class, 100, 269–270
 event dispatchers, 96. *See also* dispatching events
 event flow, 275
 event handlers
 defined, 96
 naming, 104, 285
 passing data when calling, 99–100
 sending event objects to, 101–103
 for system events, 107–111
 event handling
 with ActionScript, 98–99
 example, 97–98
 overview, 96–97
 EventLab application, 274
 event listeners, 96, 179, 275, 288
 event objects. *See also* events
 generic, 101
 inspecting, 104–107
 using data from, 100–103
 events, 100–107. *See also* event handling; event objects
 communicating with, 445–448
 data binding as, 179–181
 declaring, 267–269
 defined, 100
 dispatching, 178–179, 183, 263–267
 drag initiator, 315
 drop target, 316
 inspecting, 104–107
 interpreting, 212

 listening to, 96, 179
 passing data with, 269–270
 purpose of, 80
 types of, 96, 107
 using data from, 100–103
 event subclasses, 270–274, 280
 event targets, 275
 eventText parameter, 108
 event variable, 105
 expressions
 E4X, 129, 133
 and loose coupling, 262
 maintaining, 262
 watch, 36, 117, 119, 136, 172, 191
 Expressions panel, 173, 191
 Expression Studio, 11
 Expressions view, 36, 118, 137, 191
 Extensible Application Markup Language (XAML), 11

F

factory methods, 150
 false/true values, 151–152, 163
 FAQs (frequently asked questions), xxi
 FedEx shipment tracking application, 8
 fill property, 408
 filterCollection() method, 258
 filtering, 202–203
 filterOrganic() method, 203
 findAny() method, 200
 findFirst() method, 199–200
 findLast() method, 200
 Flash Builder
 adjusting layout of views in, 26
 basic vocabulary for, 18
 creating projects in, 18–24
 debugger, 34–41
 deleting projects in, 41–42
 displaying code line numbers in, 27
 enabling conditional breakpoints in, 40
 help/support, xxi

- importing projects into, 41
- laying out interface in, 45
- naming projects in, 18–19
- and object-oriented best practices, 95
- purpose of, 14, 17
- using Auto-Generation with, 181–183
- versions, 14
- viewing/correcting errors in, 32–33, 34–41
- workbench, 24–27
- Flash Catalyst, 14
- Flash Debug perspective, 26
- Flash Platform, 12–15
- Flash Player
 - and AIR, 12
 - and application height/width, 24
 - compilation process, 14
 - evolution of, 12
 - how it works internally, 439
 - popularity of, 12
 - and runtime-loaded CSS files, 395
 - sandboxes, 122–123
 - as single-threaded virtual machine, 439
 - versions, 13–14
- Flash Professional, 12–13
- Flash Text Engine, 81
- Flex
 - application architecture, 212–213
 - application development, 18–24
 - applications (*See* applications)
 - basic vocabulary for, 18
 - Community Help, xx–xxii
 - compiler, 176–183
 - as component-based development model, 207
 - components, 421–422
 - event-based programming model, 95, 96–97
 - getting started with, xii–xiii, 17–43
 - home page, xxii
 - key technologies, 13–14
 - language tags, 23
 - and object-oriented programming, 41
 - positioning of components in, 46
 - purpose of, xii, 13
 - resources, xx–xxii
 - Spark components, 23
 - versions, xii, 13
 - working with view states in, 63–70
- FlexGrocer application
 - adding events to, 95
 - building checkout process for, 335–358
 - controlling view states for, 68–70
 - creating list of products for, 185–191
 - creating/organizing files for, 18–24
 - customizing with skins, 399–417 (*See also* skins)
 - customizing with styles, 375–397 (*See also* styles)
 - defining product section for, 57–58
 - displaying images in, 81–84
 - displaying/managing data for, 144–150
 - externalizing data in, 114–116
 - formatting currency for prices in, 364–369
 - implementing checkout process for, 333–358
 - implementing drag/drop in, 326–331
 - improving architecture for, 213–221
 - laying out interface for, 50–58
 - manipulating shopping cart data for, 159–167
 - modifying, 30–33
 - overview, xv
 - Product data structure for, 141
 - providing product/category information for, 230–238
 - refactoring, 73–75, 101–103
 - running, 28–29
 - validating postal codes for, 369–372
 - visual shopping cart for, 169 (*See also* shopping carts)
 - website for, xvi
 - working with view states for, 63–70
- flexgrocer.com, xvi
- Flex Properties view, 82–83
- Flex SDK, xii, xiii, 14, 19, 23, 80
- flow-driven programming model, 96
- fonts, embedded, 388, 390
- for loops, 162–163, 164–165
- Formatter classes, 364–368
- formatters
 - for displaying currency, 364–368, 443
 - examples of, 362–363
 - purpose of, 361, 363
- Form container, 47, 81, 89–92
- form fields, 89
- FormHeading component, 91
- FormItem component, 90
- FormLayout object, 47
- forms, 89, 91
- Forums, Adobe, xxii
- FreeBSD, 11
- frequently asked questions (FAQs), xxi
- functions. *See also* specific functions
 - parameters vs. arguments for, 147
 - private vs. protected, 267
 - scope of, 102
 - vs. methods, 144
- </fx:Declarations> tag, 29
- FXG Graphics, 404–405
- FXG specification, 407
- <fx:Metadata> tag, 267–268
- <fx:Model> tag, 88
- fx namespace, 23
- FXP files, 34, 41
- <fx:Script> block, 99
- <fx:Style> tag, 382–383
- <fx:XML> tag, 149, 151

G

Generate Getter/Setter dialog box, 182, 249, 428
 getItemAt() method, 192–193
 getItemInCart() method, 163
 getters/setters, 177–178, 181–183, 249, 428
 Google Maps, 10
 “go to definition” shortcut, 171
 graphical elements, 46, 404–405
 graphics editing programs, 404–405
 graphics property, 404–405
 greater-than sign (>), 30
 GridColumn class, 296–299
 Group container, 46, 245
 Group tag, 48–49

H

handleCreationComplete() method, 109, 149, 152
 handleViewCartClick() method, 102, 103, 104
 hasFormat() method, 322
 HGroup container, 72–73
 HorizontalLayout object, 47
 HostComponent metadata, 433
 HTML (Hypertext Markup Language)
 and drag/drop operations, 5
 latest version of, 10
 limitations of, 4–5, 10
 as page-based architecture, 4
 HTML 5 specification, 10
 HTTP (Hypertext Transport Protocol), 5, 7
 HTTPServices
 accessing data retrieved from, 121–122
 creating, 120, 230–231
 retrieving XML data via, 124–126
 Hypertext Markup Language.
 See HTML
 Hypertext Transport Protocol (HTTP), 5, 7

I

IDataRenderer interface, 246, 247
 IDE (integrated development environment), 14
 ID selectors, 382, 392
 IFill interface, 408
 IFormatter interface, 362
 IFrames, 9
 Image control, 81–84
 images
 displaying, 81–84
 editing, 404–405
 loading at runtime, 81
 loading at start of application, 84
 scaling, 83
 implicit getters/setters, 177–178, 181–183
 importing
 classes, 109–110, 215
 projects, 41, 68
 import statements, 110, 157, 304
 includeIn property, 67, 87, 334
 inheritable style properties, 381
 inheritance
 hierarchy, 263
 and invalidateProperties() method, 440
 and protected functions, 267
 style, 376, 381, 384
 inline editors, 296–299
 inline item renderers, 303–304
 inline styles, 379–381
 instance methods, 151
 instances
 Array, 184, 194
 bindable, 194
 DataGroup, 242
 event, 276, 277
 Label, 193
 List, 242, 313
 Object, 184
 ObjectProxy, 118, 135
 validator, 371
 vs. properties, 49
 XML, 184

integers, unsigned, 155
 integrated development environment (IDE), 14
 interactivity, 5, 63
 interfaces, 199. *See also* UI
 Internet
 applications (*See* web applications)
 dot-com boom, 7
 explosive growth of, 4
 security issues, 122–123
 invalidateProperties() method, 440
 isItemInCart() method, 163
 italics, xvii
 itemEditor property, 297
 item editors, 296–299
 itemRenderer class, 246–251, 256
 item renderers
 for displaying products, 299–302
 how they work, 246
 implementing, 246–251
 inline, 296–299, 303–304
 items. *See also* products
 adding to shopping cart, 159–161
 displaying based on category, 257–258
 dragging to shopping cart, 311, 326–331
 finding in shopping cart, 161–163
 updating quantity of, 161, 164–166
 IT organizations, 8
 IValidator interface, 363
 IViewCursor interface, 198, 199

J

Java, xiv
 Java applets, 9
 JavaFX, 10
 JavaScript, 5, 9
 Java Virtual Machine, 10
 just-in-time (JIT) compilation, 14

K

keyboard shortcuts, xvii

L

Label component, 378

Label controls, 57, 60, 80

LabelFunction property

displaying subtotal with,
305–308

purpose of, 242

using with lists, 242–245

Label property, 49

Label tag, 29

LastResult property, 121–122

layout anchors, 59

layout objects, 46, 47–48

Layout property, 48

layouts, 45–63. *See also* containers
adding elements in Design view,
53–56

combining containers and, 48

constraint-based, 55, 58–63

for e-commerce application,
50–58

purpose of, 46

starting in Source view, 51–53

lessons

directory structure for, xviii

list of, xiv–xv

minimum system requirements,
xix

standard elements used in,
xvi–xvii

Letterbox scale mode, 83, 84

Linear Gradient class, 408–409

line breaks, 31

Linux, 11

list-based components, 321

List controls. *See also* lists

displaying category data with,
137–139

dragging/dropping between,
313–315

dragging/dropping between
DataGrid and, 315–321

populating with dataset,
242–245

using LabelFunction with,
242–245

lists. *See also* List controls

formatting data in, 244–245

responding to user's choice
from, 257–259

virtualization with, 255–257

Local History feature, 31–32

loops, 162–163, 164–165

loosely coupled architecture,
262–263

M

Mac OS-based systems

manifest files for, 23

and Silverlight, 11

system requirements, xix

Macromedia, xii, 12, 13

mainframes, 4

Major League Baseball application, 8

manifest files, 23

menu commands, xvii

messaging, 15

metadata tags, 267

methods. *See also* specific methods

creating objects with, 150–153

defining, 143

DragManager class, 322

DragSource class, 322

factory, 150

instance, 151

overriding, 271

parameters vs. arguments for, 147

static, 151

vs. functions, 144

microcomputers, 4

Microsoft

Expression Studio, 11

Silverlight, 11

minHeight attribute, 31

minWidth attribute, 31

MLB.TV Media Player, 8

models, 212

model-view-controller (MVC)
architecture, 88, 212–213

Moonlight, 11

mouseDown event, 315

MouseEvent properties, 104–107

mouseMove event, 315

mouseout event, 87

mouseover event, 87

multiDisplay() function, 244

MVC (model-view-controller)
architecture, 88, 212–213

mx components, 20

mx.formatters package, 362

MXML

case-sensitivity of, 21

class instances vs. properties in,
49–50

compiling, 149

components, 208–213, 420
(*See also* components)

creating applications in, 18–24

creating classes in, 149

decoding tags in, 49–50

formatting rules/standards, 30

item renderers, 303–304

purpose of, 13

MXMLC compiler, 395

MXML Component dialog box, 214

mx.validators package, 363–364

N

name collision, 147, 148

@namespace declaration, 389

namespaces

fx namespace, 23

how CSS defines, 389

s namespace, 23

Spark, 23, 387, 389

styles and, 383–384

navigation system

importance of, 333

and loose coupling, 262–263

using states as basis for, 334,
337–345

nested quotes, 97, 99

.NET, xiv, 11
 new keyword, 151
 New Package command, 221
 New State icon, 64
 newTotal variable, 164, 165
 nodes, 128, 186
 non-visual components, 230–238

O

object-oriented programming
 (OOP), xiv, 41, 95, 102, 141, 215
 ObjectProxy, 184
 objects
 building method to create,
 150–153
 classes as blueprint for, 141
 converting XML to, 117–119, 133
 data transfer, 143
 event (*See* event objects)
 OOP. *See* object-oriented
 programming
 Open Font Library, 388
 Open Perspective button, 22, 26
 Open Perspective icon, 107
 open source site, 81
 OrderEvent object, 355–358
 OrderInfo object, 335–336, 355
 Outline view, 57, 65

P

Package Explorer, 22, 25, 41, 145
 package keyword, 145
 packages, 143, 145, 221
 page-based architecture, 4, 6–7, 8
 Panel container, 46
 parameters, 147
 partAdded() method, 445–448
 partRemoved() method, 445,
 448–449
 PC manifest files, 23
 personal computers, 4
 perspectives, 26, 36, 107
 PHP, xiv
 postal-code validator, 369–372

prefix, 23
 prepare method, 300, 301
 private functions, 267
 private keyword, 158
 Problems view, 22, 32
 ProductEvent class, 280–286,
 304–305
 ProductItem components
 breaking out, 221–230
 cleaning up appearance of,
 376–377
 creating instances of, 228
 productivity, 8
 ProductList component
 creating, 282–284, 426
 styling labels in, 391
 using, 284–285
 product nodes, 186
 products. *See also* items
 adding/removing from
 shopping cart, 284–286,
 304–305
 creating, 189
 displaying names of, 148–149
 filtering based on category,
 257–259
 keeping track of shopping cart,
 154–159
 product section, 57–58
 programmatic graphics, 404–405
 programming languages, xiv
 Project menu, 28
 projects. *See also* applications
 creating, 18–24
 deleting, 41–42
 importing, 41, 68
 naming, 18–19, 41
 overwriting, 41
 viewing, 21–22
 viewing errors in, 32–33
 properties. *See also* specific
 properties
 creating, 215
 declaring, 143
 vs. class instances, 49
 Properties panel, 53, 57

protected functions, 267
 proxies
 array, 184–185
 drag, 312
 pseudo-selectors, 382, 393–394
 public properties, 146–147

Q

question mark (?), 146
 Quick Fix tool, 146, 180
 quotation marks ("), 68, 97

R

RadialGradient class, 408
 redundancy, 71
 refactoring, 71–75
 applications, 73–75, 101–103
 benefits of, 71–72
 defined, 71
 to search with cursor, 198–201
 ShoppingCart class, 287–288
 ShoppingCartItem class,
 204–205
 remote XML data, 110–139
 dynamic, 133–137
 embedded XML, 114–116
 searching with E4X, 127–133
 security issues, 122–123
 XMLListCollection, 137–139
 XML loaded at runtime, 119–123
 Remove button, 303–304
 removeItem() method, 304
 Reset Perspective command, 53
 ResultEvents, 101
 result handler, 122
 Resume button, 38
 return types, 99, 102
 RIAs (rich Internet applications),
 3–15
 advantages of, 7–8
 and drag/drop technique, 311
 examples of excellent, 8
 functions of, 5–6
 goals of, 6–7

- technology choices, 8–14
 - vs. traditional web applications, 6–7
 - RichEditableText control, 81
 - rich Internet applications. *See* RIAs
 - RichText control, 80, 81, 86, 109
 - rollover event, 85–87
 - root nodes, 128–129
 - Run button, 22, 28, 29
 - runtime
 - changing CSS at, 395–397
 - changing skins at, 448
 - loading images at, 81, 84
 - styling at, 395–397
 - XML loaded at, 119–123
- S**
- sandboxes, 122–123
 - `<s:Application>` tag, 22–23, 31
 - satellite navigation system, 262–263
 - `<s:BasicLayout/>` tag, 52
 - scalar values, 98
 - Scale Mode menu, 83, 84
 - scope, 102
 - Script blocks, 99, 102
 - scroll bars, 24
 - Scroller tag, 48–49
 - scrolling content, 48–49
 - SDK (software development kit), xii, xiii, 14, 19, 23, 80
 - searches
 - array, 246
 - with cursor, 198–201
 - descendant, 190
 - XML (with E4X), 127–133
 - security issues, 122–123
 - security sandboxes, 122–123
 - selectability, 255
 - `selectedIndex` property, 255
 - `selectedItem` property, 255
 - self-closing tags, 31, 51
 - `send()` method, 120, 126
 - servers, 4. *See also* client/server applications
 - server-side languages, xiv
 - server-side objects, 6
 - server technologies, 15
 - service-oriented architecture (SOA), 5
 - `setStyle()` method, 381
 - setters/getters, 177–178, 181–183, 249, 428
 - `<s:FormHeading>` tag, 89
 - `<s:FormItem>` tag, 89
 - `<s:Form>` tag, 89
 - shipment tracking application, 8
 - ShoppingCart class. *See also* shopping carts
 - building, 154–159
 - refactoring, 287–288
 - replacing Array in, 194
 - ShoppingCartItem class, 154–159, 204–205
 - shopping carts
 - adding items to, 63, 159–161, 284–286
 - displaying with DataGrid, 291, 292–308
 - dragging items to, 311, 326–331
 - finding items in, 161–163
 - formatting list data for, 244–245
 - keeping track of items in, 154–159
 - manipulating data in, 157, 159–167
 - removing items from, 201–202, 284–286
 - updating quantity of items in, 164–166
 - updating totals in, 287–288
 - ShoppingList component, 425–452
 - adding functionality to, 439–450
 - checking functionality of, 429–431
 - choosing base class for, 426
 - creating class for, 427–429
 - creating custom renderer for, 450–452
 - creating skin for, 434–438
 - defining interface for, 425
 - specifying skin requirements for, 432–434
 - ShoppingView class, 251–253
 - Show Line Numbers command, 27
 - Show View command, 25
 - Silverlight, 11
 - simple controls, 79–92
 - linking data structure to, 88–89
 - overview, 80–81
 - purpose of, 80
 - tools for laying out, 79
 - using Form container to lay out, 89–92
 - ways of using, 79, 80
 - SkinnableComponent class, 421–422, 445
 - SkinnableContainer, 46
 - SkinnableDataContainer, 47
 - skinning, 46, 432
 - SkinPart metadata, 432
 - skin parts, 401, 434
 - skins, 399–417
 - changing at runtime, 448
 - creating, for Application component, 413–416, 434–438
 - creating, for FlexGrocer button, 400–404
 - creating, for ShoppingList component, 434–438
 - creating renderer for, 450–452
 - customizing button states with, 407–413
 - errors, 401
 - purpose of, 376
 - relationship between states and, 404–413
 - role of, in Spark components, 400–404
 - vs. styles, 376
 - SkinState metadata, 432–433
 - `<s:Label>` tag, 29
 - slash (/), 30, 31
 - `<s:List>` control, 137–138
 - s namespace, 23
 - SOA (service-oriented architecture), 5
 - software development kit (SDK), xii, xiii, 14, 19, 23, 80

- software upgrades, 4
 - SolidColor class, 408
 - someColor property, 171–172
 - SortField objects, 194–197
 - Source button, 22
 - Source view, 25, 51–53
 - Spark classes, 23
 - Spark components
 - and embedded fonts, 390
 - namespace for, 23, 387, 389
 - role of skins in, 400–404, 413
 - vs. MX components, 20
 - spark.formatters package, 362
 - Spark namespace, 23, 387, 389
 - spark.validators package, 363–364
 - <s:states> tag, 337
 - stateless protocols, 7
 - states
 - controlling, 68–70
 - creating, 64
 - creating navigation structure using, 334, 337–345
 - maintaining, 7
 - naming, 68
 - relationship between skins and, 404–413
 - setting properties for, 67
 - state selectors, 393–394
 - static methods, 151
 - Step Into button, 37, 40
 - Step Over button, 37, 40
 - stretch scale mode, 83
 - style inheritance, 381
 - StyleManager, 396–397
 - styleName property, 383
 - style properties, 379, 381
 - styles, 375–397. *See also* skins
 - assigning multiple, 382, 397
 - complete list of, 378
 - CSS inheritance for, 381
 - overriding, 397
 - purpose of, 376
 - at runtime, 395–397
 - setting with CSS files, 386–394
 - setting with <fx:Style> tag, 382–383
 - setting with setStyle() method, 381
 - setting with tag attributes, 379–381
 - vs. skins, 376
 - ways of applying, 377–378
 - subclasses, 270–274, 280
 - subtotals, 156, 204, 305–308
 - Support pages, xxi
 - SWF files, 29, 395–396
 - system events, 96, 107–111
 - system requirements, xix
- T**
- tag attributes, setting styles via, 379–381
 - tags
 - choosing attributes for, 30
 - Form container, 89
 - selecting, 30
 - self-closing, 31, 51
 - target phase, event flow, 275
 - target property, 100, 105
 - targets, event, 275
 - text
 - controls, 80–81
 - displaying blocks of, 85–87
 - styles for manipulating, 378–379
 - TextArea component, 80, 108
 - TextInput control, 80, 91, 247
 - Text Layout Framework (TLF), 81
 - this keyword, 105, 147
 - tight coupling, 262
 - TileLayout object, 47
 - timestamp property, 270–274
 - timestamps, 269
 - TLF (Text Layout Framework), 81
 - toString() method, 148–149, 150, 151, 156, 160
 - total property, 165, 166
 - trace() method, 149, 150, 153, 156
 - training centers, Adobe, xxii
 - Training from the Source* series, xviii
 - transfer objects, 143
 - true/false values, 151–152, 163
 - tutorials, xxi
 - two-way bindings, 334
 - type property, 100
 - type selectors, 382, 383–385
- U**
- UI (user interface), 45–76
 - arranging elements in, 58
 - drag-and-drop technique, 311 (*See also* drag/drop operations)
 - dynamic, 71
 - for e-commerce application, 45
 - HTML limitations, 4–5
 - laying out, 50–58
 - tools for creating, 11
 - UIComponent class, 208, 263, 421
 - underscore (_), 177
 - unique identifier (UUID), 41
 - uni tRPCResult() handler, 122
 - Universal Resource Identifier (URI), 23
 - unsigned integers, 155
 - updateItem() method, 164
 - URI (Universal Resource Identifier), 23
 - URLs, 23, 120
 - UserAcknowledgeEvent class, 270–274
 - user events, 96, 97, 107
 - user frustration level, 8
 - user input forms, 334
 - user interface. *See* UI
 - users, collecting information from, 89
 - UUID (unique identifier), 41

V

Validator classes, 363, 364, 369–372

validators

- for checking postal codes, 369–372
- examples of, 363–364
- purpose of, 361, 363

value objects, 143–150, 153

values

- attribute, 98
- Boolean, 151–152, 163
- scalar, 98
- setting/reading, 177
- true/false, 151–152, 163

variables

- controlling, 363
- integer, 35
- name collision among, 147
- naming, 147
- in RIAs, 7
- showing current state of, 36

Variables view, 36, 37–38, 105

vector graphics, 404–405

VerticalLayout object, 47, 57

VGroup container, 72–73, 85

video publishing, 15

View Cart buttons, 101–103

views

- adjusting layout of, 26
- displaying list of, 25
- grouping, 36
- in MVC architecture, 212, 213
(*See also* specific views)
- opening/closing, 25

view states, 63–70

- controlling, 68–70
- creating, 63–67
- defined, 63
- naming, 68

virtualization

- implementing, 254–255
- with List class, 255–257
- power of, 255
- purpose of, 253–254

visual components, 213–230, 274

void return type, 99, 102

W

WarningDialog application, 264–267

watch expressions, 36, 117, 119, 136, 172, 191

Web 2.0, xii

web applications. *See also* applications

- connectivity issues, 6
- and drag/drop technique, 311
- and event-based programming, 95
- evolution of, 4–6
- flow for traditional, 6–7
- inefficiencies of, 6
- maintaining state in, 7

web browsers, 4, 5

Web Services, 6

Window menu, 25

Windows-based systems

- manifest files for, 23
- and Silverlight, 11
- system requirements, xix

Windows Presentation Foundation (WPF), 11

workbench, 24–27

workflow engines, 15

World Wide Web, xiv. *See also* Internet

WPF (Windows Presentation Foundation), 11

X

XAML (Extensible Application Markup Language), 11

XML

- ActionScript support for, 127
- and AJAX, 9
- and code completion, 150
- converting to objects, 117–119, 133
- embedded, 114–119
- and Flex, 13
- formatting rules/standards, 30
- loaded at runtime, 119–123
- namespaces, 23
- nomenclature, 23
- terminating tags in, 31
- vs. XMLList, 128

XML class, 127

XML data

- accessing returned, 121–122
- dynamic, 133–137
- remote (*See* remote XML data)
- retrieving via HTTPService, 124–126
- security issues, 122–123

XMLDocument class, 127

XMLHttpRequest, 9

XMLList, 128

XMLListCollection, 133, 135, 137–139

Z

zip-code validator, 369–372

ZipCodeValidator class, 369–372



WATCH READ CREATE

Unlimited online access to all Peachpit, Adobe Press, Apple Training and New Riders videos and books, as well as content from other leading publishers including: O'Reilly Media, Focal Press, Sams, Que, Total Training, John Wiley & Sons, Course Technology PTR, Class on Demand, VTC and more.

No time commitment or contract required!
Sign up for one month or a year.
All for \$19.99 a month

SIGN UP TODAY
peachpit.com/creativeedge

creative
edge

WHERE ARE THE LESSON FILES?

Adobe Flex 4.5 Fundamentals: Training from the Source

The print version of this title comes with a disc of lesson files. As an eBook reader, you have access to these files by following the steps below:

1. On your PC or Mac, open a web browser and go to this URL:
<http://www.peachpit.com/ebookfiles/0132788918>
2. Download the ZIP file (or files) from the web site to your hard drive.
3. Unzip the files and follow the directions for use in the Read Me included in the download.

Please note that many of our lesson materials can be very large, especially image and video files. You will be able to see the size of any file for download once you reach the URL listed above.

If you are unable to locate the files for this title by following the steps above, please email ask@peachpit.com and supply the URL from step one. Our customer service representatives will assist you as soon as possible.

Legal Notice: Peachpit Press makes no warranty or representation, either express or implied, with respect to this software, its quality, performance, merchantability, or fitness for a particular purpose. In no event will Peachpit Press, its distributors, or dealers be liable for direct, indirect, special, incidental or consequential damages arising out of the use or inability to use the software. The exclusion of implied warranties is not permitted in some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights. There may be other rights that you may have that vary from state to state.

The software and media files are copyrighted by the authors and Peachpit Press. You have the non-exclusive right to use these programs and files. You may use them on one computer at a time. You may not distribute the URL to third parties or redistribute the files over a network. You may transfer the files onto a single hard disk so long as you can prove ownership of this eBook.

You may not reverse engineer, decompile, or disassemble the software or media files. You may not modify or translate the software or media, or distribute copies of the software or media without the written consent of Peachpit Press.



JOIN THE
PEACHPIT
AFFILIATE TEAM!

You love our books and you love to share them with your colleagues and friends...why not earn some \$\$ doing it!

If you have a website, blog or even a Facebook page, you can start earning money by putting a Peachpit link on your page.

If a visitor clicks on that link and purchases something on peachpit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post an ad and we'll take care of the rest.

APPLY AND GET STARTED!

It's quick and easy to apply.

To learn more go to:

<http://www.peachpit.com/affiliates/>

*Valid for all books, eBooks and video sales at www.Peachpit.com



Peachpit