



ADOBE® COLDFUSION® 9

Application Development

web application construction kit
VOLUME 2

Ben Forta

Adobe ColdFusion 9 Web Application Construction Kit, Volume 2: Application Development

Ben Forta and Raymond Camden

with Charlie Arehart, John C. Bland II, Ken Fricklas, Paul Hastings, Mike Nimer, Sarge Sargent, and Matt Tatam

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
Peachpit is a division of Pearson Education

Copyright ©2011 by Ben Forta

Series Editors: Rebecca Freed and Karen Reichstein
Editor: Judy Ziajka
Technical Reviewer: Brian Rinaldi
Production Editor: Tracey Croom
Compositor: Maureen Forys, Happenstance Typo-O-Rama
Proofreader: Liz Welch
Indexer: Ron Strauss
Cover design: Charlene Charles-Will

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 nor Peachpit 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

Adobe, ColdFusion, ColdFusion Builder, Dreamweaver, Flash, Flash Builder, Flex, and LiveCycle are trademarks or registered trademarks of Adobe Systems, Inc., in the United States and/or other countries. All other trademarks are the property of their respective owners. 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.

ISBN 13: 978-0-321-67919-2
ISBN 10: 0-321-67919-9

This page intentionally left blank

CONTENTS AT A GLANCE

<i>Introduction</i>	<i>xxiii</i>
PART 5 Creating Functions, Tags, and Components	1
CHAPTER 22 Building User-Defined Functions	3
CHAPTER 23 Creating Custom Tags	23
CHAPTER 24 Creating Advanced ColdFusion Components	51
PART 6 ColdFusion Configuration and Performance	95
CHAPTER 25 ColdFusion Server Configuration	97
CHAPTER 26 Managing Threads	133
CHAPTER 27 Improving Performance	151
PART 7 Integrating with ColdFusion	173
CHAPTER 28 Working with PDF Files	175
CHAPTER 29 ColdFusion Image Processing	221
CHAPTER 30 Advanced ColdFusion-Powered Ajax	243
CHAPTER 31 Integrating with Adobe Flex	281
CHAPTER 32 Integrating with Flash Data Services	303
CHAPTER 33 Building ColdFusion-Powered AIR Applications	323
CHAPTER 34 Creating Presentations	335
CHAPTER 35 Full-Text Searching	349
CHAPTER 36 Event Scheduling	379
PART 8 Advanced ColdFusion Development	395
CHAPTER 37 Using Stored Procedures	397
CHAPTER 38 Working with ORM	415
CHAPTER 39 Using Regular Expressions	455
CHAPTER 40 ColdFusion Scripting	487
CHAPTER 41 Working with XML	511
CHAPTER 42 Manipulating XML with XSLT and XPath	531
PART 8 Online Chapters *	
CHAPTER 43 Manipulating XML with XSLT and XPath	E1
CHAPTER 44 Error Handling	E47
CHAPTER 45 Using the Debugger	E71
<i>Index</i>	<i>557</i>

* Pages mentioned throughout the text as online content are included after the index.

This page intentionally left blank

CONTENTS

Introduction

xxiii

PART 5	Creating Functions, Tags, and Components	1
CHAPTER 22	Building User-Defined Functions	3
	Thinking About Extending CFML	3
	Functions Turn Input into Output	4
	Building Your First UDF	4
	Basic Steps	5
	Using the Function	6
	UDF Tag Syntax	6
	Using Local Variables	8
	Where to Save Your UDFs	10
	Creating Libraries of Related UDFs	13
	Designing the UDF Library	13
	Putting the UDF Library to Use	16
	Creating General-Purpose UDFs	18
	Things to Consider	18
	Writing the <code>SimpleJavaScriptFunctions</code> Library	19
	Sharing UDF Libraries with Others	21
CHAPTER 23	Creating Custom Tags	23
	Easy, Powerful Extensibility	23
	Introducing CFML Custom Tags	23
	The Basic Custom Tag Idea	24
	How to Use Custom Tags	24
	How to “Install” a Custom Tag	24
	Using Custom Tags	25
	Changing the Custom Tag Search Path	26
	Placing Custom Tags in the Current Directory	27
	Specifying Locations with <code>Application.cfc</code>	28
	Writing Custom Tags That Display Information	28
	Writing Your First Custom Tag	28
	Introducing the <code>ATTRIBUTES</code> Scope	29
	Making Attributes Optional or Required	31
	Using <code><cfparam></code> to Establish Default Values	31
	Custom Tags That Process Data	34
	Introducing the <code>CALLER</code> Scope	34
	Returning Variables to the Calling Template	34
	Variable Names as Tag Attributes	37
	Using <code><cfparam></code> with <code>type="variableName"</code>	37
	Setting a Variable Dynamically	38

Additional Custom Tag Topics	40
Passing Attributes with <code>attributeCollection</code>	40
Working with <code><cfimport></code>	41
Advanced Custom Tags	42
Paired Custom Tags	43
The Execution Cycle	43
The Concept of <code>GeneratedContent</code>	45
Custom Tags That May Be Called as Paired or Empty	46
Using <code>thisTag.HasEndTag</code>	46
Using <code>cfexit</code> to Control Custom Tag Processing Flow	47
Nested Custom Tags	48
Where to Go from Here	49
CHAPTER 24 Creating Advanced ColdFusion Components	51
Review of ColdFusion Components	51
About ColdFusion Components	51
The Two Types of Components	53
Simple CFCs	54
Structure of a CFC File	54
CFCs as Groups of Functions	56
Using the CFC in ColdFusion Pages	58
Separating Logic from Presentation	61
Introspection and HINTs	62
<code>cfDump</code> and the <code>GetMetaData()</code> Function	63
Accessing a CFC via a URL	65
Getting Raw Data from a ColdFusion Component via a URL	65
Accessing a CFC via a Form	66
Type Checking in Components	67
Components That Hold Instance Data	68
Introducing the <code>THIS</code> Scope	69
Instance Data CFC Example	70
Storing CFCs in the <code>APPLICATION</code> Scope	75
Storing CFCs in the <code>SESSION</code> Scope	76
Instance Data as Properties	76
Keeping Your Properties Clean: Getters and Setters	79
Implicit Getters and Setters	80
Initializing Components	81
CFCs, Shared Scopes, and Locking	82
Working with Inheritance	84
Overriding Properties and Methods	85
Using the <code>SUPER</code> Scope	86
Defining Interfaces	87
Implementing Security	88
Implementing Access Control	88
Implementing Role-Based Security in CFCs	89

Using the OnMissingMethod Method	91
Distributed CFCs and Serialization	92
PART 6 ColdFusion Configuration and Performance	95
CHAPTER 25 ColdFusion Server Configuration	97
The ColdFusion Administrator	97
The Administrator Home Page	97
Server Settings	98
The Settings Page	98
The Request Tuning Page	104
The Caching Page	107
The Client Variables Page	109
The Memory Variables Page	110
The Mappings Page	111
The Mail Page	112
The Charting Page	114
The Font Management Page	115
The Document Page	116
The Java and JVM Page	116
Settings Summary	117
Data and Services	118
The Data Sources Page	118
The ColdFusion Collections Page	118
The Verity K2 Server Page	119
The Solr Server Page	119
Migrate Verity Collections	119
The Web Services Page	119
The Flex Integration Page	120
Debugging and Logging	120
The Debug Output Settings Page	120
The Debugging IP Addresses Page	122
The Debugger Settings Page	123
The Logging Settings Page	123
The Log Files Page	124
The Scheduled Tasks Page	124
The System Probes Page	125
The Code Analyzer Page	125
The License Scanner Page	126
Server Monitoring	126
The Server Monitor Page	126
Extensions	127
The Java Applets Page	127
The CFX Tags Page	127

The Custom Tag Paths Page	128
The CORBA Connectors Page	128
Event Gateways	129
The Settings Page	129
The Gateway Types Page	129
The Gateway Instances Page	130
Security	130
The Administrator Page	130
The RDS Password Page	131
The Sandbox Security Page	131
The User Manager Page	131
Packaging and Deployment	132
The ColdFusion Archives Page	132
The J2EE Archives Page	132
CHAPTER 26 Managing Threads	133
Using the <code><cfthread></code> Tag	134
Scripting the <code><cfthread></code> tag	135
Starting Threads	136
Suspending Threads	136
Ending Threads	137
Joining Threads	137
Accessing Thread Data	138
Thread Scopes	138
Monitoring and Administering Threads	144
Handling Thread Exceptions	144
Administrator Tools	148
Best Practices	149
CHAPTER 27 Improving Performance	151
Options in the ColdFusion Administrator	151
Improving Query Performance with Caching	152
Understanding Query Caching	152
Using Cached Queries	154
Refreshing Cached Queries Programmatically	156
Limiting the Number of Cached Queries	158
Controlling the Number of Records Fetched at Once	158
Caching Page Output	159
Introducing the <code><cfcache></code> Tag	159
Client-Side Page Caching	159
Server-Side Page Caching	161
ColdFusion-Optimized Caching	162
Flushing the Page Cache	163
Caching Page Fragments	164
Caching Data	164
Using the Caching Functions	166

Inspecting the Cache	167
Digging Even Deeper	168
Controlling White Space	168
Understanding the Issue	169
Automatic White-Space Control	169
Suppressing White-Space Output with <code><cfsilent></code>	170
Suppressing Specific White Space with <code><cfsetting></code>	170
PART 7 Integrating with ColdFusion	173
CHAPTER 28 Working with PDF Files	175
Using <code>cfpdf</code>	175
Creating PDF Files	176
Reading PDF Files	177
Merging PDF Files	178
Using the <code>pages</code> Attribute	179
Deleting Pages	180
Creating Thumbnails	180
Extracting Images	182
Creating Watermarks	182
Extracting Text	184
Optimizing PDFs	187
Creating PDF Portfolios	189
Manipulating Headers and Footers	189
Protecting PDF Files with Passwords, Permissions, and Encryption	192
Adding Encryption	194
Using <code>DDX</code>	196
Creating a Simple <code>DDX</code> Processing Application	197
Adding a Table of Contents	198
Adding Headers and Footers	200
Adding Watermarks	202
Using Style Profiles	203
Extracting PDF Text	205
Working with PDF Forms	208
Populating PDF Forms	208
Submitting PDF Forms	213
Embedding PDF Forms	216
Creating PDF Files from Microsoft Office Documents	216
Converting Microsoft Word Documents to PDF Files	216
Creating the Résumé Manager Application	217
CHAPTER 29 ColdFusion Image Processing	221
Introducing <code><cfimage></code>	221
Manipulating Images with <code><cfimage></code>	223
Resizing an Image	223
Adding Borders	227

Controlling JPEG Quality and Drawing Text	228
Adding a Watermark	231
Adding a CAPTCHA Test	232
Exploring Other ColdFusion Image Manipulation Techniques	235
Creating and Drawing Images with ColdFusion	237
In Summary	241
CHAPTER 30 Advanced ColdFusion-Powered Ajax	243
Ajax-Based Layout Controls	243
Working with Tabs	247
Working with Accordions	249
Working with Pods	250
Working with Windows	251
Working with Menus	252
Adding Tooltips	255
Working with Trees	256
Working with Message Boxes	257
Handling Multiple File Uploads	258
Working with Maps	260
Playing Videos	261
Displaying a Progress Bar	262
Dynamically Populating Ajax Controls	263
Using Bindings to Load Content	266
Working with <CFAjaxProxy>	271
AjaxProxy as a Binding	274
Working with JSON	274
Special Considerations with ColdFusion Ajax Applications	276
Importing JavaScript Ajax Libraries	276
Running Functions on Page Load	278
Defining JavaScript Functions in Loaded Pages	278
Debugging Ajax	278
Security Options	279
CHAPTER 31 Integrating with Adobe Flex	281
Understanding the Relationship Between ColdFusion and Flex	281
ColdFusion-Powered Flex	284
No More Request and Response	284
Asynchronous Requests and Callbacks	285
Session Scope	285
Data Translation	286
Returning Simple Objects	286
Returning Value Objects	287
ColdFusion Data Wizards	290
Calling the Server (RPC)	290
Result and Fault Handlers	291

Invoking a CFC Method	292
AsyncToken	292
ColdFusion Flex Services	293
Data Messaging	294
Messaging in Flex	294
Messaging in ColdFusion	295
Data Management	295
ColdFusion and Flex Configuration Settings	296
ColdFusion Administrator Settings	296
services-config.xml	297
Debugging	299
Console Output	299
Flash Builder and ColdFusion Line Debuggers	302
Network Monitors	302
CHAPTER 32 Integrating with Flash Data Services	303
Configuring Flash Builder for Flash Remoting	303
Configuring the Flash Builder -services Compiler Argument	304
Flash Remoting Application	304
Adding Best-Practice Improvements to the Data Grid	308
Adding an Edit Form to the Data Grid	309
Data Management Application	311
Configuring the ColdFusion Data Push Destination	312
Synchronizing the Data Grid	314
Adding Conflict Resolution to the Form	318
Messaging Application	318
Configuring the ColdFusion Data Messaging Destination	319
Creating an Event Gateway	319
Conclusion	321
CHAPTER 33 Building ColdFusion-Powered AIR Applications	323
Creating AIR Clients for ColdFusion Applications	323
Comparison with LCDS Data Management	323
Configuring the Flash Builder Project	324
Building the ColdFusion Service CFC	325
The fetch Method	325
The sync Method	326
Building the Value Objects	328
Creating the ColdFusion Value Object	328
Creating the ActionScript Value Object	329
Initializing syncManager	329
Loading Data	331
Editing and Saving Data	332
Synchronizing Data with the Server	333
Managing Conflicts	333

CHAPTER 34 Creating Presentations	335
Presenting <cfpresentation>	335
Setting Play and Control Options	340
Embedding Content	340
Styling Your Presentation	341
Adding Presenters	343
Saving Your Presentation	345
Converting from PowerPoint	346
CHAPTER 35 Full-Text Searching	349
Getting to Know Solr	349
Searching for Different Types of Files with Solr	350
Creating a Search Tool for Your Documents	350
Understanding Collections	351
Creating a New Collection	351
Indexing the Collection	352
Creating a Search Interface	355
Indexing SQL Data	359
Searching Without Solr	360
Indexing Your Table Data: Solr to the Rescue	361
Understanding Solr and Your Table's Key Values	370
Searching on More Than One Collection	370
Improving Search Results	370
Maintaining Collections	374
Repopulating Your Solr Collection	374
Administering Collections with <cfcollection>	375
Optimizing a Solr Collection	376
Deleting a Collection	376
Creating a Collection Programmatically	377
CHAPTER 36 Event Scheduling	379
ColdFusion Event Scheduling Concepts	379
ColdFusion Scheduling Versus Other Kinds of Scheduling	380
Creating, Modifying, and Deleting Scheduled Tasks	381
Administering Scheduled Tasks	381
Creating Scheduled Tasks with the ColdFusion Administrator	382
Running, Modifying, and Deleting Scheduled Tasks	384
Creating, Modifying, and Deleting Tasks Using <CFSCHEDULE>	385
Scheduling Application Examples	389
Creating a Scheduled Task for a POP3 Application	389
Building an Automatic Promotional Email Application	390
Scheduling Updates to a Solr Collection	392

PART 8	Advanced ColdFusion Development	395
CHAPTER 37	Using Stored Procedures	397
	Why Use Stored Procedures?	397
	Calling Stored Procedures from ColdFusion Templates	398
	Two Ways to Execute Stored Procedures	398
	Using the <code><cfstoredproc></code> Tag	399
	Stored Procedures That Return Record Sets	401
	Stored Procedures That Take Parameters and Return Status Codes	403
	Calling Procedures with <code><cfquery></code> Instead of <code><cfstoredproc></code>	410
CHAPTER 38	Working with ORM	415
	Welcome to ORM	415
	ColdFusion and Hibernate	416
	Getting Started with ORM	416
	Working with Persistent CFCs	417
	Working with Properties	418
	Working with Entities	420
	Putting It Together	429
	Hibernate and Sessions	431
	Working with Relationships	432
	One-to-Many and Many-to-One Relationships	433
	Many-to-Many Relationship	437
	One-to-One Relationship	443
	Enhancing Your Entities	448
	Searching with ORM	448
	Lazy Is Good!	452
	Digging Deeper into Hibernate	453
CHAPTER 39	Using Regular Expressions	455
	Introducing Regular Expressions	455
	What Are Regular Expressions?	455
	RegEx Support in ColdFusion	458
	Where Can You Use Regular Expressions?	458
	Using Regular Expressions in ColdFusion	459
	Finding Matches with <code>reFind()</code>	459
	Working with Subexpressions	462
	Working with Multiple Matches	465
	Replacing Text using <code>reReplace()</code>	465
	Altering Text with Backreferences	469
	Some Convenient RegEx UDFs	472
	Using a RegEx Testing Page	472
	Crafting Your Own Regular Expressions	473
	Understanding Literals and Metacharacters	473
	Introducing the Cast of Metacharacters	474
	Metacharacters 101: Character Classes	474

Metacharacters 102: Quantifiers	477
Metacharacters 201: Alternation	481
Metacharacters 202: Word Boundaries	481
Metacharacters 203: String Anchors	481
Metacharacters 301: Match Modifiers	484
Metacharacters 302: Lookahead Matching	485
Metacharacters 303: Backreferences Redux	486
Metacharacters 304: Escape Sequences	486
CHAPTER 40 ColdFusion Scripting	487
A New Era in CFML Scripting	487
Times Have Changed	487
Now CFML Has Changed	488
What Is <code><cfscript></code> ?	488
Differences Between <code><cfscript></code> and JavaScript	490
Implementing <code><cfscript></code>	490
Implementing CFCs in Script	491
Replacing Tags with Script	492
Using Script Statements	494
Using Script Functions	500
Using Script Functions (CFCs)	502
Defining Functions in Script	505
Making Variables Private to the Function	506
Improved Support for Function Typing	506
Exception Handling in Script	507
Common Problems and Solutions	508
A Note About the <code>{}</code> Symbols	509
“Could not find the ColdFusion Component or Interface”	509
Conclusion	510
CHAPTER 41 Working with XML	511
XML Document Structure	511
Elements and Their Attributes	513
Naming Conventions	514
Reading and Creating XML Documents	514
Reading an XML File Using <code>xmlParse()</code>	514
Creating XML Documents Using <code>CFXML</code>	516
Creating XML Documents Using <code>xmlNew()</code>	518
Accessing XML Elements and Attributes	519
Using Special Characters in XML	521
Entity References	522
CDATA Sections	522
XML Namespaces	523
The Default Namespace	525

When to Use Namespaces	526
Validating XML	526
DTDs	527
XML Schemas	527
Validating XML in ColdFusion	528
More XML Resources	530
CHAPTER 42 Manipulating XML with XSLT and XPath	531
Understanding XPath	531
Example: A CD Collection	531
XPath Syntax	533
Using <code>xmlSearch()</code> to Retrieve an Array of Nodes	535
Transforming XML into Content by Using XSLT	536
Creating a Basic Transformation	537
Performing the Transformation by Using <code>xmlTransform()</code>	542
Ignoring Nodes in the Hierarchy	542
Creating a More Complex Transformation	543
More XPath and XSLT Resources	555
PART 8 Online Chapters *	
CHAPTER 43 ColdFusion and Globalization	E1
Why Go Global?	E1
What Is Globalization?	E2
Globalization Terminology	E2
Dancing the Globalization Jig	E3
Going Global	E3
Locales	E3
Character Encoding	E14
Resource Bundles	E16
Addresses	E24
Date/Time	E25
Calendars	E25
Calendar CFC Use	E28
Time Zones	E30
Databases	E31
Display	E33
Text Searching with Solr	E35
What's New in ColdFusion Internationalization	E36
Relevant ColdFusion Tags and Functions	E37
Better G11N Practices	E40
What Not to Do	E40
Monolingual or Multilingual Web Sites	E41
Locale Stickiness	E42
HTML	E42

CFML	E43
Resource Bundles	E43
Just Use Unicode	E45
When ColdFusion Isn't Enough	E45
CHAPTER 44 Error Handling	E47
Catching Errors as They Occur	E47
What Is an Exception?	E47
Introducing <code><cftry></code> and <code><cfcatch></code>	E48
Basic Exception Handling	E49
A Typical Scenario	E49
A Basic Template, Without Exception Handling	E50
Adding <code><cftry></code> and <code><cfcatch></code>	E52
Understanding What Caused the Error	E53
Writing Templates That Work Around Errors	E56
Working Around a Failed Query	E56
Writing Templates That Recover from Errors	E59
Nesting <code><cftry></code> Blocks	E61
Deciding Not to Handle an Exception	E64
Exceptions and the Notion of Bubbling Up	E65
Using <code><cfrethrow></code>	E65
Throwing and Catching Your Own Errors	E68
Introducing <code><cfthrow></code>	E68
Throwing Custom Exceptions	E69
Handling Exceptions with Scripting	E70
CHAPTER 45 Using the Debugger	E71
Overview	E72
Traditional Forms of Debugging	E72
Introducing Step Debugging	E72
Configuring ColdFusion and the Debugger	E73
Configuring the ColdFusion Administrator	E74
Configuring ColdFusion Builder	E76
Using the Debugger	E80
Setting a Breakpoint	E82
Starting a Debugging Session	E82
Browsing a Page to Be Debugged	E85
Stepping Through Code	E88
Observing Variables and Expressions	E91
Observing the Debug Output Buffer	E93
Stopping the Debugger	E94
Other Features	E95
<i>Index</i>	557

* Pages mentioned throughout the text as online content are included after the index.

IN THIS INTRODUCTION

Who Should Use This Book xxiii

How to Use This Book xxiii

The Web Site xxvi

Introduction

Who Should Use This Book

This book is written for anyone who wants to create cutting-edge Web-based applications.

If you are a Webmaster or Web page designer and want to create dynamic, data-driven Web pages, this book is for you. If you are an experienced database administrator who wants to take advantage of the Web to publish or collect data, this book is for you, too. If you are starting out creating your Web presence but know you want to serve more than just static information, this book will help get you there. If you have used ColdFusion before and want to learn what's new in ColdFusion 9, this book is also for you. Even if you are an experienced ColdFusion user, this book provides you with invaluable tips and tricks and also serves as the definitive ColdFusion developer's reference.

This book teaches you how to create real-world applications that solve real-world problems. Along the way, you acquire all the skills you need to design, implement, test, and roll out world-class applications.

How to Use This Book

This is the ninth edition of *ColdFusion Web Application Construction Kit*, and what started as a single volume a decade ago has had to grow to three volumes to adequately cover ColdFusion 9. The books are organized as follows:

- **Volume 1**—*Adobe ColdFusion 9 Web Application Construction Kit, Volume 1: Getting Started (ISBN 0-321-66034-X)* contains Chapters 1 through 21 and is targeted at beginning ColdFusion developers.
- **Volume 2**—*Adobe ColdFusion 9 Web Application Construction Kit, Volume 2: Application Development (ISBN 0-321-67919-9)* contains Chapters 22 through 45

and covers the ColdFusion features and language elements that are used by most ColdFusion developers most of the time. (Chapters 43, 44, and 45 are online.)

- **Volume 3—*Adobe ColdFusion 9 Web Application Construction Kit, Volume 3: Advanced Application Development (ISBN 0-321-67920-2)*** contains Chapters 46 through 71 and covers the more advanced ColdFusion functionality, including extensibility features, as well as security and management features that will be of interest primarily to those responsible for larger and more critical applications.

These books are designed to serve two different, but complementary, purposes.

First, as the books used by most ColdFusion developers, they are a complete tutorial covering everything you need to know to harness ColdFusion's power. As such, the books are divided into parts, or sections, and each section introduces new topics building on what has been discussed in prior sections. Ideally, you will work through these sections in order, starting with ColdFusion basics and then moving on to advanced topics. This is especially true for the first two books.

Second, the books are invaluable desktop references. The appendixes and accompanying Web site contain reference chapters that will be of use to you while developing ColdFusion applications. Those reference chapters are cross-referenced to the appropriate tutorial sections, so that step-by-step information is always readily available to you.

The following describes the contents of *Adobe ColdFusion 9 Web Application Construction Kit, Volume 2: Application Development*.

Part V: Creating Functions, Tags, and Components

Chapter 22, “Building User-Defined Functions,” introduces the `<cffunction>` tag and explains how it can (and should) be used to extend the CFML language.

Chapter 23, “Creating Custom Tags,” teaches you how to write your own tags to extend the CFML language—tags written in CFML itself.

Chapter 24, “Creating Advanced ColdFusion Components,” continues exploring ColdFusion Components by introducing advanced topics, including persistence, encapsulation, and inheritance.

Part VI: ColdFusion Configuration and Performance

Chapter 25, “ColdFusion Server Configuration,” revisits the ColdFusion Administrator, this time explaining every option and feature, while providing tips, tricks, and hints you can use to tweak your ColdFusion server.

Chapter 26, “Managing Threads,” explains asynchronous development and how to use multi-threaded processing to improve application performance.

Developers are always looking for ways to tweak their code, squeezing a bit more performance wherever possible. Chapter 27, “Improving Performance,” provides tips, tricks, and techniques you can use to create applications that will always be snappy and responsive.

Part VII: Integrating with ColdFusion

Adobe PDF files are the standard for high-fidelity document distribution and online forms processing, and ColdFusion features extensive PDF integration, as explained in Chapter 28, “Working with PDF Files.”

Chapter 29, “ColdFusion Image Processing,” teaches you how to read, write, and manipulate image files using ColdFusion tags and functions.

Chapter 30, “Advanced ColdFusion-Powered Ajax,” continues to explore Ajax user interface controls and concepts.

Chapter 31, “Integrating with Adobe Flex,” introduces the basics of ColdFusion-powered Flex applications.

Chapter 32, “Integrating with Flash Data Services,” discusses ColdFusion and Flash, exploring Flash Remoting, LiveCycle Data Services, Blaze DS, and more.

Chapter 33, “Building ColdFusion-Powered AIR Applications,” teaches you how to use ColdFusion to build desktop applications, including applications that can be taken offline.

Chapter 34, “Creating Presentations,” teaches you how to use ColdFusion to build dynamic Acrobat Connect presentations.

Chapter 35, “Full-Text Searching,” introduces the Apache Solr search engine. Solr provides a mechanism that performs full-text searches of all types of data. The Solr engine is bundled with the ColdFusion Application Server, and the `<cfindex>` and `<cfsearch>` tags provide full access to Solr indexes from within your applications.

Chapter 36, “Event Scheduling,” teaches you how to create tasks that run automatically and at timed intervals. You also learn how to dynamically generate static HTML pages using ColdFusion’s scheduling technology.

Part VIII: Advanced ColdFusion Development

Chapter 37, “Using Stored Procedures,” takes advanced SQL one step further by teaching you how to create stored procedures and how to integrate them into your ColdFusion applications.

Object Relational Mapping, or ORM, provides a powerful new way to build data-driven applications, with an emphasis on rapid development and simplified ongoing maintenance. Chapter 38, “Working with ORM,” introduces this new ColdFusion 9 capability and explains how to fully use this powerful Hibernate-based technology.

Chapter 39, “Using Regular Expressions,” introduces the powerful and flexible world of regular expression manipulation and processing. Regular expressions allow you to perform incredibly sophisticated and powerful string manipulations with simple one-line statements. ColdFusion supports the use of regular expressions in both find and replace functions.

Chapter 40, “ColdFusion Scripting,” introduces the `<CFSCRIPT>` tag and language, which can be used to replace blocks of CFML code with a cleaner and more concise script-based syntax.

<CFSCRIPT> can also be used to create ColdFusion Components and user-defined functions, both of which are explained in this chapter, too.

Extensible Markup Language (XML) has become the most important means of exchanging and sharing data and services, and your ColdFusion applications can interact with XML data quite easily. Chapter 41, “Working with XML,” explains what XML is and how to use it in your ColdFusion code.

Chapter 42, “Manipulating XML with XSLT and XPath,” explains how to apply XSL transformations to XML data, as well as how to extract data from an XML document using XPath expressions.

The Internet is a global community, and multilingual and localized applications are becoming increasingly important. Chapter 43, “ColdFusion and Globalization” (online)*, explains how to build these applications in ColdFusion to attract an international audience.

Chapter 44, “Error Handling” (online)*, teaches you how to create applications that can both report errors and handle error conditions gracefully. You learn how to apply the <cftry> and <cfcatch> tags (and their supporting tags) and how to use these as part of a complete error-handling strategy.

Chapter 45, “Using the Debugger” (online)*, explores the ColdFusion Builder debugger and offers tips and tricks on how to best use this tool.

The Web Site

The book’s accompanying Web site contains everything you need to start writing ColdFusion applications, including:

- Links to obtain ColdFusion 9
- Links to obtain Adobe ColdFusion Builder
- Source code and databases for all the examples in this book
- Electronic versions of some chapters
- An errata sheet, should one be required
- An online discussion forum

The book Web page is at <http://www.fortta.com/books/0321679199/>.

And with that, turn the page and start reading. In no time, you’ll be creating powerful applications powered by ColdFusion 9.

* Pages mentioned throughout the text as online content are included after the index.

This page intentionally left blank

CHAPTER 24

Creating Advanced ColdFusion Components

IN THIS CHAPTER

Review of ColdFusion Components	51
Simple CFCs	54
Introspection and HINTs	62
Accessing a CFC via a URL	65
Type Checking in Components	67
Components That Hold Instance Data	68
Working with Inheritance	84
Defining Interfaces	87
Implementing Security	88
Using the <code>OnMissingMethod</code> Method	91
Distributed CFCs and Serialization	92

Review of ColdFusion Components

An important part of ColdFusion is its ColdFusion Components (CFCs) framework. Think of the CFC framework as a special way to combine key concepts from custom tags and user-defined functions into *objects*. These objects might represent concepts (such as individual films or actors), or they might represent processes (such as searching, creating special files, or validating credit card numbers).

I covered the basics of ColdFusion Components in Chapter 11, “The Basics of Structured Development,” in *Adobe ColdFusion 9 Web Application Construction Kit, Volume 1: Getting Started*, but I’ll review them here.

About ColdFusion Components

You can think of CFCs as a structured, formalized variation on custom tags. The CFC framework gently forces developers to work in a more systematic way. If you choose to use the CFC framework for parts of your application, you will find yourself thinking about those aspects in a slightly more structured, better organized way. Because CFCs are more structured, the code is generally very easy to follow and troubleshoot. Think of the CFC framework as a way to write smart code, guiding you as a developer to adopt sensible practices.

But the most dramatic benefit is that the structured nature of CFCs makes it possible for ColdFusion to look into your CFC code and find the important elements, such as what functions you have included in the CFC and what each function’s arguments are. This knowledge allows ColdFusion Builder to act as a kind of interpreter between your CFC and other types of applications, such as Dreamweaver, Flash, and Web Services. If you want them to, these components become part of a larger world of interconnected clients and servers, rather than only being a part of your ColdFusion code.

CFCs Can Be Called in Many Different Ways

This chapter and the previous one have been all about making it easier to reuse the code that you and other developers write. CFCs take the notion of code reuse to a whole new level, by making it easy to reuse your code not only within ColdFusion but in other types of applications as well. Components can be called directly in ColdFusion pages, but the functions in them can also be called directly from external URLs, like Web pages that return data instead of HTML. Because of this, CFCs both can provide functionality to ColdFusion pages similarly to custom tags and user-defined functions (UDFs) and can also be called directly from Flash, from Ajax code in Web browsers, and as Web Services from other applications not on the same machine as the CFCs.

In other words, if you like the idea of reusing code, you'll love the CFC framework even more than the UDF and custom tag frameworks.

CFCs Are Object-Oriented Tools

Depending on your background, you may be familiar with object-oriented programming (OOP). Whether you know OOP or not, CFCs give you the most important real-world benefits of object-oriented programming without getting too complicated—exactly what you would expect from ColdFusion.

Without getting too deeply into the specifics, you can think of object-oriented programming as a general programming philosophy. The philosophy basically says that most of the concepts in an application represent objects in the real world and should be treated as such. Some objects, like films or merchandise for sale, might be physical. Others, like expense records or individual merchandise orders, might be more conceptual but still easy to imagine as objects—or objectified, like many of Orange Whip Studios' better-looking actors.

ColdFusion's CFC framework is based on these object-oriented ideas:

- **Classes.** In traditional object-oriented programming, the notion of a class is extremely important. For our purposes, just think of an object class as a type of object, or a *thing*. For instance, Orange Whip Studios has made many films during its proud history. If you think of each individual film as an object, then it follows that you can consider the general notion of a film (as opposed to a particular film) as a class. CFCs themselves are the classes in ColdFusion.
- **Methods.** In the object-oriented world, each type of object (that is, each class) will have a few *methods*. Methods are functions that have been conceptually attached to a class. A method represents something an object can do. For instance, think about a car as an object. A car has to start, change gears, stop, accelerate, and so on. So, a corresponding object class called car might have methods named `Car.start()`, `Car.shift()`, `Car.avoidPedestrian()`, and so on.
- **Instances.** If there is a class of object called Film, then you also need a word to refer to each individual film the studio makes. In the OOP world, this is described as an *instance*. Each individual film is an instance of the class called Film. Each instance of an object

usually has some information associated with it, called its *instance data*. For example, Film A has its own title and stars. Film B and Film C have different titles and different stars.

- **Properties.** Most real-world objects have properties that make them unique, or at least distinguish them from other objects of the same type. For instance, a real-world car has properties such as its color, make, model, engine size, number of doors, license plate and vehicle identification numbers, and so on. At any given moment, it might have other properties such as whether it is currently running, who is currently driving it, and how much gas is in the tank. If you're talking about films, the properties might be the film's title, the director, how many screens it is currently shown on, or whether it is going to be released straight to video. Properties are just variables that belong to the object (class), and they are generally stored as instance data.
- **Inheritance.** In the real world, objects have various types—cars are a type of motorized vehicle, which is a type of conveyance, while a bicycle is also a conveyance. ColdFusion's CFC framework allows you to define an order of inheritance, where you can have properties and methods that are shared between various kinds of objects share the high-level stuff and then implement more specific versions with custom features. In our car example, you'd have a conveyance that might define number of wheels. Bicycle, motor vehicles, and skateboards are all types (called *subclasses*) of conveyances. Cars are a subclass of motor vehicle (as are trucks), and electric cars are a subclass of cars. I'll talk about this in more detail later in the chapter.

The Two Types of Components

Most CFCs fall into two broad categories: *static* components and *instance-based* components.

Static Components

I'll use the term *static* to refer to any component where it doesn't make sense to create individual instances of the component. These contain methods (functions, remember?) but no data that hangs around after a function runs. Often you can think of such components as *services* that are constantly listening for and answering requests. For instance, if you were creating a film-searching component that made it easy to search the current list of films, you probably wouldn't need to create multiple copies of the film-searching component.

Static components are kind of like Santa Claus, the Wizard of Oz, or your father—only one of each exists. You just go to that one and make your request.

Instance-Based Components

Other components represent ideas where it is very important to create individual instances of a component. For instance, consider a CFC called `ShoppingCart`, which represents a user's shopping cart on your site. Many different shopping carts exist in the world at any given time (one for each user). Therefore, you need to create a fresh instance of the `ShoppingCart` CFC for each new Web visitor, or perhaps each new Web session. You would expect most of the CFC's methods to return

different results for each instance, depending on the contents of each user’s cart. Instance-based components contain properties as well as functions, which define the differences between each instance of the component and the other instances.

Simple CFCs

The best news about CFCs is that there is really very little to learn about them. For the most part, you just write functions in much the same way that you learned in the previous chapter.

When you want to use user-defined functions (UDFs) in general, you have to include the files that contain the functions on every page that runs them. Frequently you’ll create a library file that contains just the functions you need—perhaps a file called `utilityfunctions.cfm` and include it on the page.

Simple static CFCs are just a different way to call these functions—one that doesn’t require you to explicitly include them and instead lets you call them similarly to custom tags.

Structure of a CFC File

Each ColdFusion component is saved in its own file, with a `.cfc` extension. Except for one new tag, `<cfcomponent>`, everything in the file is ordinary CFML code. With the `.cfc` extension instead of `.cfm`, the ColdFusion server can easily detect which files represent CFC components.

Introducing the `<cfcomponent>` Tag

The `<cfcomponent>` tag doesn’t have any required attributes, so in its simplest use, you can just wrap opening and closing `<cfcomponent>` tags around everything else your CFC file contains (mainly `<cffunction>` blocks). That said, you can use two optional attributes, `hint` and `displayName`, to make your CFC file more self-describing (see Table 24.1).

If you provide these optional attributes, ColdFusion and Dreamweaver can automatically show `hint` and `displayName` in various places to make life easier for you and the other developers who might be using the component.

Table 24.1 `<cfcomponent>` Tag Syntax

ATTRIBUTE	DESCRIPTION
<code>hint</code>	Optional. What your component does, in plain English (or whatever language you choose, of course). I recommend that you provide this attribute.
<code>displayName</code>	Optional. An alternative, friendlier phrasing of the component’s name. Make the component’s actual name (that is, the file name) as self-describing as possible, rather than relying on the <code>displayName</code> to make its purpose clear.
<code>output</code>	Optional. See <code>output</code> under <code><cffunction></code> below. Only affects any code not inside a <code><cffunction></code> .

NOTE

As you will soon see, the `<cffunction>` and `<cfargument>` tags also have `hint` and `displayName` attributes. Each aspect of a CFC that someone would need to know about to actually use it can be described more completely within the component code itself.

Using `<cffunction>` to Create Methods

The biggest part of a CFC is the ColdFusion code you write for each of the CFC's methods (functions). To create a component's methods, you use the `<cffunction>` tag the same way you learned in Chapter 23, "Creating Custom Tags." If the method has any required or optional arguments, you use the `<cfargument>` tag, again as shown in Chapter 23.

The `<cffunction>` and `<cfargument>` tags each take a few additional attributes that Chapter 23 didn't discuss because they are relevant only for CFCs. The most important new attributes are `hint` and `displayName`, which all the CFC-related tags have in common. Tables 24.2 and 24.3 summarize all `<cffunction>` and `<cfargument>` attributes.

Table 24.2 `<cffunction>` Syntax for CFC Methods

ATTRIBUTE	DESCRIPTION
<code>name</code>	Required. The name of the function (method), as discussed in Chapter 23.
<code>hint</code>	Optional. A description of the method.
<code>displayName</code>	Optional.
<code>returnType</code>	Optional.
<code>returnFormat</code>	Optional. The format in which the data should be returned when accessed remotely. By default, all data is returned in WDDX format, unless <code>returnType</code> is XML. You can specify WDDX, JSON (for JSON format, used by Ajax), or plain (for no formatting).
<code>access</code>	Optional. This attribute defines where your method can be used. See the "Implementing Security" section below for more information.
<code>roles</code>	Optional. A list of security roles or user groups that should be able to use the method. Again, see the "Implementing Security" section below for more information.
<code>output</code>	Optional. If <code>false</code> , acts like the entire function is within a <code><cfsilent></code> tag. If <code>true</code> , acts like the entire function is within a <code><cfoutput></code> tag. If not set, acts normal; variables being output must be in <code><cfoutput></code> tags.

NOTE

The valid data types you can provide for `returnType` are `any`, `array`, `binary`, `component`, `Boolean`, `date`, `guid`, `numeric`, `query`, `string`, `struct`, `uuid`, `variableName`, and `xml`. If the method isn't going to return a value at all, use `returnType="void"`. If the method is going to return an instance of another component, you can provide that component's name (the file name without the `.cfc`) as the `returnType` value.

Table 24.3 <cfargument> Syntax for CFC Method Arguments

ATTRIBUTE	SYNTAX
name	Required. The name of the argument.
hint	An explanation of the argument's purpose. Like the HINT attribute for <cfcomponent> and <cffunction>, this description will be visible in Dreamweaver to make life easier for you and other developers. It is also included in the automatic documentation that ColdFusion produces for your components.
displayName	Optional, friendly name.
type	Optional. The data type of the argument. You can use any of the values mentioned in the note below Table 24.2 except for void.
required	Optional. Whether the argument is required.
default	Optional. A default value for the argument, if required="No".

NOTE

There is actually another CFC-related tag, called <cfproperty>. See the "Introspection and HINTs" section below.

<cfcomponent> and <cffunction> also have many other optional attributes that are discussed in the chapters on Web Services, ORM, and ActionScript.

CFCs as Groups of Functions

Let's look at a simple example of a CFC. Say you want to create a CFC called `FilmSearchCFC`, which provides a simplified way to search for films or print out the results. You like the idea of being able to reuse this component within your ColdFusion pages, instead of having to write queries over and over again. You'd also like to be able to flip a switch and have the component available to Flash Player or Web Services.

Listing 24.1 is a simple version of the `FilmSearchCFC`.

Listing 24.1 `FilmSearchCFC.cfc`—A Simple CFC

```
<!--
  Filename: FilmSearchCFC.cfc
  Author: Ken Fricklas (KF)
  Purpose: Creates FilmSearchCFC, a simple ColdFusion Component
-->

<!-- The <CFCOMPONENT> block defines the CFC --->
<!-- The filename of this file determines the CFC's name --->
<cfcomponent hint="Search and display films">

    <cffunction name="listFilms" returnType="query" output="false" access="remote"
    hint="Search for a film, and return a query with the id and title of the matching
    films.">
        <!-- Optional SearchString argument --->
        <cfargument name="searchString" required="no" default="" hint="movie title to
        search for. If not provided, returns all films.">

            <!-- var scoped variables --->
```

Listing 24.1 (CONTINUED)

```

<cfset var getFilms = "">
<!-- Run the query -->
<cfquery name="getFilms" datasource="ows">
SELECT FilmID, MovieTitle FROM Films
<!-- If a search string has been specified -->
<cfif ARGUMENTS.searchString neq "">
WHERE (MovieTitle LIKE '%#ARGUMENTS.searchString#%'
OR Summary LIKE '%#ARGUMENTS.searchString#%')
</cfif>
ORDER BY MovieTitle
</cfquery>

<!-- Return the query results -->
<cfreturn getFilms>

</cffunction>

<cffunction name="printFilms" returnType="void" access="remote" hint="Search for a
film, and display the results in an HTML table.">
<cfargument name="searchString" required="no" default="" hint="Movie title to
search for. If not provided, returns all films.">
<!-- call the local function getFilms with the argument searchString -->
<cfset var qFilms = listFilms(arguments.searchString)>
<table>
<tr><th>ID</th><th>Title</th></tr>
<cfoutput query="qFilms">
<tr><td>#qFilms.FilmID#</td><td>#qFilms.MovieTitle#</td></tr>
</cfoutput>
</table>
<!-- Return the query results -->
<cfreturn>

</cffunction>
</cfcomponent>

```

NOTE

Earlier, I explained that there are two types of components: static components, which just provide functionality, and instance-based components, which provide functionality but also hold information. This CFC is an example of a static component. You will see how to create instance-based components shortly.

NOTE

The `access` attribute is set to `remote`, so this component can be called directly from a Web browser, as you'll see later in this chapter.

This version of the CFC has two methods: `listFilms()`, which queries the database for a listing of current films, and `printFilms()`, which prints them out as an HTML table. For `listFilms()`, the query object is returned as the method's return value (this is why `returnType="query"` is used in the method's `<cffunction>` tag).

The `listFilms()` method takes one optional argument called `searchString`. If the `searchString` argument is provided, a `WHERE` clause is added to the database query so that only films with titles or summaries containing the argument string are selected. If the `searchString` isn't provided, all films are retrieved from the database and returned by the new method.

`PrintFilms()` takes the same arguments but outputs the data as an HTML table. Since it does not return a value, it returns `void` as the return type.

As you can see, building a simple component isn't much different from creating a user-defined function. Now that you've created the component, let's take a look at how to use it in your ColdFusion code.

Using the CFC in ColdFusion Pages

Once you have completed your CFC file, there are two basic ways to use the new component's methods in your ColdFusion code:

- With the `<cfinvoke>` tag, as discussed next.
- Using the new keyword (`new` in ColdFusion 9) to create and initialize the object and calling its methods using function syntax, in the form `component.methodName()`. (You can also use the `<cfobject>` tag or the `createObject()` function, although these are less used since the introduction of the `new` keyword, which is simpler and does more.)

Calling Methods with `<cfinvoke>`

The most straightforward way to call a CFC method is with the `<cfinvoke>` tag. `<cfinvoke>` makes your CFC look a lot like a custom tag. To provide values to the method's arguments, as in the optional `searchString` argument in Listing 24.1, either you can add additional attributes to `<cfinvoke>` or you can nest a `<cfinvokeargument>` tag within the `<cfinvoke>` tag. Tables 24.4 and 24.5 show the attributes supported by `<cfinvoke>` and `<cfinvokeargument>`.

Table 24.4 `<cfinvoke>` Tag Syntax

ATTRIBUTE	DESCRIPTION
<code>component</code>	The name of the component, as a string (the name of the file in which you saved the component, without the <code>.cfc</code> extension) or a component instance.
<code>method</code>	The name of the method you want to use.
<code>returnVariable</code>	A variable name in which to store whatever value the method decides to return.
(method arguments)	In addition to the <code>component</code> , <code>method</code> , and <code>returnVariable</code> attributes, you can also provide values to the method's arguments by providing them as attributes. For instance, the <code>listFilms()</code> method from Listing 24.1 has an optional argument called <code>searchString</code> . To provide a value to this argument, you could use <code>searchString="Saints"</code> or <code>searchString="#FORM.keywords#"</code> . You can also provide arguments using the separate <code><cfinvokeargument></code> tag (see Table 24.5).
<code>argumentCollection</code>	Optional. This attribute lets you provide values for the method's arguments together in a single structure. It works the same way as the <code>attributeCollection</code> attribute of the <code><cfmodule></code> tag. This is great for passing all your arguments to another function, as you'll see in the section on inheritance.

NOTE

For the `component` attribute, you can use the component name alone (that is, the file without the `.cfc` extension) if the `.cfc` file is in the same folder as the file that is using the `<cfinvoke>` tag. You can also specify a `.cfc` file in another folder, using dot notation to specify the location of the folder relative to the Web server root, where the dots represent folder names. For instance, you could use the `FilmSearchCFC` component by specifying `component="ows.24.FilmSearchCFC"`. For more information, see the ColdFusion 9 documentation.

NOTE

You can also save `.cfc` files in the special `CustomTags` folder (or its subfolders) or in a mapped folder. Specify the path from the `customtag` root or from the mapping using the dot syntax above.

Table 24.5 `<cfinvokeargument>` Tag Syntax

ATTRIBUTE	DESCRIPTION
<code>name</code>	The name of the argument as specified in the arguments of the method
<code>value</code>	The value of the argument

Listing 24.2 shows how to use `<cfinvoke>` to call the `listFilms()` method of the `FilmSearchCFC` component created in Listing 24.1.

Listing 24.2 Using `FilmSearchCFC.cfm`—Invoking a Component Method

```
<!---
  Filename: UsingFilmSearchCFC.cfm
  Author: Nate Weiss (NMW)
  Purpose: Uses the FilmSearchCFC component to display a list of films
  -->

<html>
<head><title>Film Search Example</title></head>
<body>

<!--- Invoke the ListFilms() method of the FilmSearchComponent --->
<cfparam name="FORM.keywords" default="ColdFusion">

<cfinvoke component="FilmSearchCFC" method="listFilms" searchString="#FORM.keywords#"
  returnVariable="FilmsQuery">

<!--- Now output the list of films --->
<cfoutput query="filmsQuery">
  #FilmsQuery.MovieTitle#<br>
</cfoutput>

</body>
</html>
```

First, the `<cfinvoke>` tag invokes the `listFilms()` method provided by the `FilmSearchCFC1` component. Note that the correct value to provide to `component` is the name of the component file name, but without the `.cfc` extension.

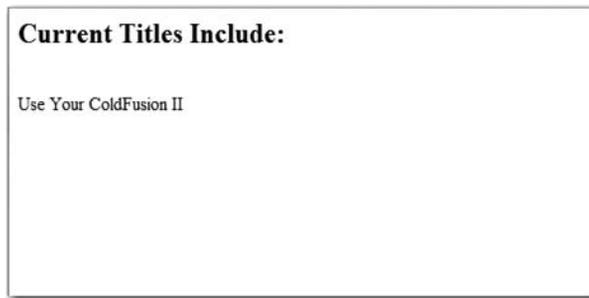
The `returnVariable` attribute has been set to `FilmsQuery`, which means that `FilmsQuery` will hold whatever value the method returns. The method in question, `listFilms()`, returns a query object as its return value. Therefore, after the `<cfinvoke>` tag executes, the rest of the example can refer to `filmsQuery` as if it were the results of a normal `<cfquery>` tag. Here, a simple `<cfoutput>` block outputs the title of each film.

We pass the argument `searchString` to the method, passing the keywords from the form (or in this case, from the `<cfparam>` tag).

The result is a simple list of film titles, as shown in Figure 24.1. Since `ColdFusion` was passed in as the `searchString`, only the single matching film is returned.

Figure 24.1

It's easy to execute a component's methods and use the results.



NOTE

You can use the `<cfinvokeargument>` tag to supply the `searchString` argument (or any other argument), instead of providing the argument as an attribute of `<cfinvoke>`. This is useful when an argument is optional and may not always be passed in because of the program logic. This is an improvement over custom tag syntax, where you have to write extra code to accomplish this.

Creating an Instance of a CFC

In the previous listing, you saw how to use the `<cfinvoke>` tag to call a CFC method. Calling methods this way isn't much different from calling a custom tag with `<cfmodule>` or calling a UDF.

It's also possible to create an instance of a CFC and then call the instance's methods. If the CFC doesn't track instance data (a shopping cart, say, or information about a particular film), there isn't much of a functional difference. It does, however, create a simpler syntax if you're going to invoke a lot of methods from a component, and you can also store the instance in a scope variable (application or session), as discussed later in this chapter.

To work with methods in this way, two steps are involved:

1. Create an instance of the CFC with the `new` keyword. You need to do this only once, since it is then in a variable that can be reused.
2. Call the method directly, using function syntax. (You can also use the `<cfinvoke>` tag, but instead of specifying the component by name, you pass the component instance variable directly to the component attribute.)

When using the new keyword, you simply set a variable to be a “new” copy of the CFC. You add parentheses to the end of the component name, as if it were a function. (You can also pass arguments to the component while creating it; I’ll talk more about this in the section on initialization later in this chapter.) For example, to use this method in the code in Listing 24.2, you’d simply replace the `<cfinvoke>` tag with the following:

```
<!-- Create an instance of the CFC -->
<cfset cfcFilmSearch = new FilmSearchCFC(>

<!-- Invoke the ListFilms() method of the CFC instance -->
<cfset filmsQuery = cfcFilmSearcher.listFilms(searchString=variables.keywords)>
```

Just as in the previous example, `filmsQuery` would now contain the query returned by the `listFilms()` method.

If the component you are initializing isn’t in the current path, you’d again use dot syntax; for example, if the code for `FilmSearchCFC` were in the path `/ows/24/FilmSearchCFC`, you would use this syntax:

```
<cfset cfcFilmSearch = new ows.24.FilmSearchCFC(>
```

You can see an example of this in action in Listing 24.3, below.

NOTE

It’s good practice to name variables that contain CFCs in a way that’s easily recognizable. In the examples below, I begin all CFC variables with `cfc`, for example `cfcFilmSearch`.

NOTE

You can also use the `<cfimport>` tag with the `path` attribute to import a directory of CFCs into the current namespace. This form of invocation will also execute the `init()` method if one is defined. `<cfimport path="ows.24.*">` would allow you to use `<cfset cfcFilmSearch = new FilmSearchCFC(>` without specifying the path. This is similar to having a mapping in the ColdFusion Administrator, but one that’s only valid for the current page. This can be great when you have multiple revisions of CFCs in different paths and are testing them.

Separating Logic from Presentation

As you can see, it’s relatively easy to create a CFC and use its methods to display data, perhaps to create some sort of master-detail interface. The process is basically first to create the CFC and then to create a normal ColdFusion page to interact with each of the methods.

When used in this fashion, the CFC is a container for *logic* (such as extraction of information from a database), leaving the normal ColdFusion pages to deal only with *presentation* of information. Many developers find it’s smart to keep a clean separation of logic and presentation while coding.

This is especially true in a team environment, where different people are working on the logic and the presentation. By keeping your interactions within databases and other logic packaged in CFCs, you can shield the people working on the presentation from the guts of your application. They can focus on making the presentation as attractive and functional as possible, without

needing to know any CFML other than `<cfinvoke>` and `<cfoutput>`. And they can easily bring up the automatically generated documentation pages for each component to stay up to date on the methods each component provides.

Introspection and HINTs

If you access your component via a Web browser, it displays all the information you have provided in your component—methods, arguments, and any documentation you have provided in the `hint` arguments in a human-readable fashion. It also shows you the return types and argument types. This is known as *introspection*. This ability to see into the information in your component is also made available in logical form and is used by Flash, Web Services, ColdFusion Builder, and Dreamweaver to make the details of your component usable from within those environments.

Assuming you installed ColdFusion on your local machine and are saving this chapter's listings in the `ows/24` folder within your Web server's document root, the URL to access a component called `FilmSearchCFC` would be as follows:

```
http://localhost/ows/24/FilmSearchCFC.cfc
```

Figure 24.2 shows the data you see in a Web browser when you navigate to this URL.

Figure 24.2
Introspection of ColdFusion component—an automatic reference page.

FilmSearch
Component FilmSearch

Search and display films

hierarchy:	WEB-INF.cftags.component FilmSearch
path:	C:\inetpub\wwwroot\WACK\FilmSearch.cfc
serializable:	Yes
properties:	
methods:	listFilms , printFilms

* - private method

listFilms

```
remote query listFilms ( searchString="" )
```

Search for a film, and return a query with the id and title of the matching films.

Output: suppressed
Parameters:
searchString: any, optional, searchString - movie title to search for. If not provided, returns all films.

printFilms

```
remote string printFilms ( searchString="" )
```

Search for a film, and display the results in an HTML table.

Output:
Parameters:
searchString: any, optional, searchString - Movie title to search for. If not provided, returns all films.

cfdump and the GetMetaData() Function

You can dump a component with `cfdump`. For example, you can dump `FilmSearchCFC` as shown here:

```
<cfobject component="FilmSearchCFC" name="cfcFilmRotation">
<cfdump var="#cfcFilmRotation#">
```

The result is shown in Figure 24.3.

Figure 24.3

The result of `<cfdump>` on a component.

The screenshot shows the output of a `cfdump` command on a component named `FilmSearchCFC`. The output is displayed in a structured HTML view with the following sections:

- component:** `FilmSearchCFC`
- METHODS:**
 - LISTFILMS:**
 - function listFilms**
 - Arguments:**

Name	Required	Type	Default
searchString	Optional	Any	[empty string]
 - ReturnType:** query
 - Roles:**
 - Access:** remote
 - Output:** false
 - DisplayName:**
 - Hint:** Search for a film, and return a query with the id and title of the matching films.
 - Description:**
 - PRINTFILMS:**
 - function printFilms**
 - Arguments:**

Name	Required	Type	Default
searchString	Optional	Any	[empty string]
 - ReturnType:** void
 - Roles:**
 - Access:** remote
 - Output:**
 - DisplayName:**
 - Hint:** Search for a film, and display the results in an HTML table.
 - Description:**

As you can see, the dump shows the component's methods and instance data. This data can be useful to your code. ColdFusion provides a means to programmatically examine an instance of a component to get this data: the `getMetaData()` function. The `getMetaData()` function returns a structure containing the same information that you can see in the HTML view of a component that `cfdump` provides.

There are two syntaxes for using the `getMetaData()` function. From outside of a component, pass the function a reference to the component object. Within a component, pass the function the component's own scope keyword `THIS`. So, for example, the code

```
<cfobject component="FilmSearchCFC" name="cfcFilmSearch">
<cfdump var="#getMetaData(cfcFilmSearch)#">
```

will produce a structure similar to that shown in Figure 24.4.

Figure 24.4

The result of the `getMetaData()` function.

getMetaData of cfcFilmSearch - struct	
EXTENDS	getMetaData of cfcFilmSearch - struct
FULLNAME	WEB-INF.cf-tags.component
NAME	WEB-INF.cf-tags.component
PATH	C:\ColdFusion9\wwwroot\WEB-INF\cf-tags\component.cfc
TYPE	component
FULLNAME	FilmSearchCFC
FUNCTIONS	getMetaData of cfcFilmSearch - array
1	getMetaData of cfcFilmSearch - struct
ACCESS	remote
HINT	Search for a film, and return a query with the id and title of the matching films.
NAME	listFilms
OUTPUT	false
PARAMETERS	getMetaData of cfcFilmSearch - array
1	getMetaData of cfcFilmSearch - struct
DEFAULT	[empty string]
HINT	movie title to search for, if not provided, returns all films.
NAME	searchString
REQUIRED	no
RETURNTYPE	query
2	getMetaData of cfcFilmSearch - struct

With this data, you could produce component HTML documentation in whatever form you wish simply by accessing the information in the structure. This approach can be useful when you want to check the properties of a component that's been passed into a function or verify whether a method is implemented in it. This specification is demonstrated in Listing 24.3.

Listing 24.3 `getMetaData.cfm`—Display Data Using `getMetaData()`

```
<!---
  getMetaData.cfm
  Demonstrate use of getMetaData() function
  --->
<!--- instantiate the FilmSearchCFC object into cfcFilmSearch --->
<cfset cfcFilmSearch = new FilmSearchCFC(>
<!--- now get the metadata, into the ourMetaData function --->
<cfset ourMetaData = getMetaData(cfcFilmSearch)>

<cfoutput>
<!--- Show the displayName and size; we could also show the hint,
  path, etc. --->
<h3>Welcome to the #ourMetaData.Name!</h3>
Enjoy our #arrayLen(ourMetaData.functions)# functions:
<ul>
<!--- loop through and show each function's name and hint; could also show
  parameters array, etc. but let's keep it simple. --->
<cfloop index="thisFunction" from="1" to="#arrayLen(ourMetaData.functions)#">
<li>#ourMetaData.functions[thisFunction].Name# - #ourMetaData.
functions[thisFunction].Hint#</li>
</cfloop>
</ul>
</cfoutput>
```

Accessing a CFC via a URL

You have seen how to use CFC methods in your `.cfm` pages using the `<cfinvoke>` and `<cfobject>` tags. It's also possible to access methods directly with a Web browser.

NOTE

I recommend that you use CFCs by invoking their methods within a `.cfm` page (using `<cfinvoke>` or the `<cfscript>` method syntax), as you have seen already, rather than having browsers visit the CFC's methods directly. This keeps the separation of functionality and presentation clean. If you do decide to have your CFCs accessed directly via a URL, keep the parts of the code that output HTML in separate methods, as the example in this section does.

To use one of the component's methods, just add a URL parameter named `method` to the example in the previous section, where the value of the parameter is the name of the method you want to call. You can also pass any arguments on the URL. For instance, to use the method called `ProduceFilmListHTML`, passing the `searchString` value of `ColdFusion`, you would visit this URL with your browser:

```
http://localhost/ows/24/FilmSearchCFC.cfm?method=printFilms&searchString=ColdFusion
```

NOTE

It is possible to access a method via a URL only if the `<cffunction>` block that creates the method contains an `access="remote"` attribute. If you try to use the URL to access a method that has a different `access` level (including the default value `public`), ColdFusion will display an error message.

To provide values for multiple arguments, just provide the appropriate number of name-value pairs, always using the name of the argument on the left side of the equals (=) sign and the value of the argument on the right side of the equals sign.

NOTE

If the value of the argument might contain special characters such as spaces or slashes, you need to escape the value with ColdFusion's `URLEncodedFormat()` function. This is the case for any URL parameter, not just for CFCs. In fact, it's the case for any Web application environment, not just ColdFusion.

If you need to provide non-simple arguments such as arrays or structures, you can do so by creating a structure that contains all of your arguments (similar to creating a structure to pass to the `attributeCollection` attribute of the `<cfmodule>` tag), using the `<cfwddx>` tag to convert the structure to a WDDX packet and then passing the packet as a single URL parameter called `argumentCollection`. Or, if you are accessing the CFC via a form, you can provide such a packet as a form field named `argumentCollection`.

NOTE

In general, methods that generate HTML or other output with `<cfoutput>` should not also produce a return value. In other words, you generally shouldn't use `<cfoutput>` and `<cfreturn>` within the same method.

Getting Raw Data from a ColdFusion Component via a URL

The example I just used calls the `printFilms` method in the component, which returns the data in an HTML table. Formatted data isn't useful if you want to return your data to a program running on another machine or if you want to consume the data from within JavaScript running on a Web

page. Fortunately, ColdFusion does something that makes these cases simple. Figure 24.5 shows the result of calling the `listFilms` method directly from a Web browser using this URL:

```
http://localhost/ows/24/FilmSearchCFC.cfc?method=listFilms&searchString=ColdFusion
```

Figure 24.5
A WDDX packet returned by running a component remotely.

```
<wddxPacket version='1.0'><header/><data><recordset rowCount='1' fieldNames='FILMID,MOVIEITITLE' type='coldfusion.sql.QueryTable'><field name='FILMID'><number>19.0</number></field><field name='MOVIEITITLE'><string>Use Your ColdFusion</string></field></recordset></data></wddxPacket>
```

This is a WDDX packet, which is an XML representation of the data. This data can be consumed via JavaScript libraries, by a ColdFusion page, or by many other languages via a WDDX interpreter.

More often these days, however, you would like to return the data in JSON format, which is what Ajax Web applications most often consume. To return JSON, all you have to do is modify the definition of the function that is returning the data by setting the `returnFormat` attribute to `JSON`, as shown in the following example for the `listFilms` method:

```
<cffunction name="listFilms" returnType="query" output="false" access="remote"
hint="Search for a film, and return a query with the id and title of the matching
films." returnFormat="JSON">
```

The result is what you see in Figure 24.6—a JSON packet that is consumable by Ajax applications. Talk about making it easy!

Figure 24.6
The JSON response returned by component.

```
{ "COLUMNS": ["FILMID", "MOVIEITITLE"], "DATA": [[19, "Use Your ColdFusion"] ] ] }
```

NOTE

Listing 24.1 contains one logic method and one presentation method. They are both included in the same CFC file. If you wanted, you could create a separate CFC for the presentation method. You would just use the `<cfinvoke>` tag within the presentation CFC to call the logic methods.

Accessing a CFC via a Form

It is also possible to access a method directly from a browser using a form. Conceptually, this is very similar to accessing a method via a URL, as discussed above in “Accessing a CFC via a URL.” Just use the URL for the `.cfc` file as the form’s action, along with the desired method name. Then add form fields for each argument that you want to pass to the method when the form

is submitted. For example, the following snippet would create a simple search form, which, when submitted, would cause a list of matching films to appear:

```
<cfform action="FilmSearchCFC.cfc?method=PrintFilms">
  <input name="searchString">
  <input type="Submit" value="Search">
</cfform>
```

NOTE

Again, the method must use `access="remote"` in its `<cffunction>` tag. Otherwise, it can't be accessed directly over the Internet via a form or a URL.

Type Checking in Components

Methods in CFCs can return types through the use of the `returnType` attribute of `<cffunction>`. Consider this example:

```
<cffunction name="listFilms" returnType="query" output="false">
```

Here, the method must return a variable with a data type of `query`. Any other return type would cause an error. For example, it might make sense to return a value of the Boolean `false` because no valid query could be returned, but that would throw an error. Instead, you'd want to return an empty query or throw a custom error.

You can also specify data types in your arguments for methods. In any `<cfargument>`, you can specify the type that your method must return (just as with `<cfparam>`). This specification can prevent you from having to create a lot of custom error-handling code in your application to check the data types of arguments passed in, and it also helps in introspection. In addition, the `<cfproperty>` tag allows you to document variables and define their types for subsequent self-documentation (more on this in the next section).

NOTE

The data type attributes of `<cffunction>` and `<cfargument>` are required when creating Web Services (see Chapter 59, "Creating and Consuming Web Services," in *Adobe ColdFusion 9 Web Application Construction Kit, Volume 3: Advanced Application Development*, for more information).

Table 24.6 lists the allowed data types.

Table 24.6 Type Values Used for `returnType` (`<cffunction>`) and `type` (`<cfargument>`, `<cfproperty>`)

TYPE	DESCRIPTION
Any	Can be any type.
Array	ColdFusion array complex data type.
Binary	String of ones and zeros.
Boolean	Can be 1, 0, true, false, yes, or no.

Table 24.6 (CONTINUED)

TYPE	DESCRIPTION
Date	Any value that can be parsed into a date. Note that POP dates (see the <code>ParseDateTime()</code> function) and time zones are not accepted, but simple timestamps and ODBC-formatted dates and times are accepted.
GUID	The argument must be a UUID or GUID of the form <code>xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx</code> where each <code>x</code> is a character representing a hexadecimal number (0–9, A–F).
Numeric	Integer or float.
Query	ColdFusion query result set.
String	ColdFusion string simple data type.
Struct	ColdFusion struct complex data type.
UUID	The argument must be a ColdFusion UUID of the form <code>xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx</code> where each <code>x</code> is a character representing a hexadecimal number (0–9, A–F).
variableName	A string formatted according to ColdFusion variable naming conventions (a letter, followed by any number of alphanumeric characters or underscores).
Void	Does not return a value.

If anything else is specified as a return type, ColdFusion processes it as returning a component for which properties have been defined. This technique allows components to define complex types for Web Services. Chapter 58, “Using Server-Side HTTP and FTP,” in Volume 3, discusses this feature in depth. Typically, though, the standard data types will suffice.

Components That Hold Instance Data

The ColdFusion Components discussed so far in this chapter (the `FilmSearchCFC` and `FilmDataCFC` examples) have both been *static* components, meaning they don’t hold any instance data. That is, although you can create an instance of a component with `<cfobject>` before using it, there really isn’t any need to do so. One instance of a component isn’t going to behave any differently from any other instance, so it’s fine to simply call the CFC’s methods directly.

If you create components that hold instance data, though, each instance of the component lives on its own and has its own memory in which to store information. If your component is about films, each instance might be an individual film, and the instance data might be the film’s title, budget, gross receipts, or even critics’ reviews. If your component is about shopping carts, each instance of the component would represent a separate user’s shopping cart, and the instance data would be the cart’s contents.

This section will explain how to create this type of component.

Introducing the THIS Scope

The CFC framework sets aside a special variable scope called `THIS`, which stands for *this instance* of a component. You can think of the word `THIS` as meaning “this film” or “this shopping cart” or “this object,” depending on what you intend your component to represent.

The THIS Scope Represents an Instance

The `THIS` scope is similar in its function to the `SESSION` scope you learned about in Chapter 19, “Working with Sessions,” in Volume 1, except that instead of being a place to store information that will be remembered for the duration of a user’s session, `THIS` is a place to store information that will be remembered for as long as a particular instance of a component continues to exist.

As an example, consider a fictional CFC called `ParrotCFC`. The idea behind the component is that each instance of the component represents one parrot. Each instance of the component needs to have a name, an age, a gender, a wingspan, a favorite word or cracker, and so on. This kind of information is exactly what the `THIS` scope was designed for. Your CFC code just needs to set variables in the `THIS` scope (perhaps `THIS.favoriteWord` or `THIS.wingSpan`) to remember these values. ColdFusion will keep each component’s variables separate.

Steps in the THIS Process

Here are the steps involved:

1. Create the CFC file. Within the file, use the `THIS` scope as the component’s personal memory space, keeping in mind that each instance of the component (that is, each parrot) will get its own copy of the `THIS` scope for its own use.
2. In your ColdFusion pages, create an instance of the CFC with `new` before you use any of the component’s methods. If you want the instance to live longer than the current page request, you can place the instance in the `SESSION` or `APPLICATION` scope.
3. Now go ahead and use the instance’s methods with the `<cfinvoke>` tag as you learned in previous examples. Make sure that you specify the instance (that is, the individual parrot) as the component attribute of the `<cfinvoke>` tag, rather than as the name of the CFC. Alternatively, call the methods using function syntax.

In this scenario, each individual instance of the `ParrotCFC` has a life of its own. The `<cfobject>` tag is what makes a particular parrot come to life. The `THIS` scope automatically maintains the parrot’s characteristics.

Extending the metaphor, if the parrot is the pet of one of your Web users, you can make the parrot follow the user around by having it live in the user’s `SESSION` scope. Or if the parrot doesn’t belong to a particular person but instead belongs to your application as a whole (perhaps the parrot is your site’s mascot), you could have it live in the `APPLICATION` scope. Or you might have a bunch of parrots that are looking for owners. You could keep these parrots (each one an instance of the `ParrotCFC` component) in an array in the `APPLICATION` scope. When a user wants to take one of the parrots home as a pet, you could move the parrot out of the array and into the `SESSION` scope.

Okay, that's enough about parrots. The idea here is to think of a CFC as an independent thing or object with its own properties. You store individual instances of the object in the APPLICATION or SESSION scope if you want it to remain in memory for a period of time, or just leave it in the normal scope if you need the instance to live only for the current page request.

NOTE

By definition, a component that doesn't refer to the THIS scope at all in its methods doesn't need to be instantiated with `<cfoobject>` before calling its methods and can therefore be considered a static component. Any component that does use the THIS scope internally probably needs to be instantiated to function properly.

Instance Data CFC Example

Let's look at a simple example of a CFC that holds instance data. The component is called `FilmRotationCFC`, and its purpose is to keep track of a featured film.

Designing `FilmRotationCFC`

To demonstrate the use of multiple methods within an instantiated component, the `FilmRotationCFC` component will contain the methods listed in Table 24.7.

Table 24.7 Methods Provided by `FilmRotationCFC`

METHOD	DESCRIPTION
<code>currentFilmID()</code>	Returns the ID number of the currently featured film. Because this method uses <code>access="Private"</code> , it can only be used internally within the <code>FilmRotationCFC</code> .
<code>isFilmNeedingRotation()</code>	Returns TRUE if the current film has been featured for more than the amount of time specified as the rotation interval (5 seconds by default). Returns FALSE if the current film should be left as is for now. This is a private method that can only be used internally.
<code>rotateFilm()</code>	Rotates the currently featured film if it has been featured for more than the amount of time specified as the rotation interval (5 seconds by default). Internally, this method calls <code>isFilmNeedingRotation()</code> to find out if the current film has expired. If so, it sets the current film to be the next film in the rotation.
<code>getCurrentFilmID()</code>	Rotates the current movie (if appropriate) and then returns the currently featured film. Internally, this function calls <code>rotateFilm()</code> and then returns the value of <code>currentFilmID()</code> . This is a public method.
<code>getCurrentFilmData()</code>	Returns the title, summary, and other information about the currently featured film. Internally, this function calls <code>getCurrentFilmID()</code> and then returns the information provided by the <code>GetFilmData()</code> method of the <code>FilmDataCFC2</code> component. This method is included mainly to show how to call one component's methods from another component.
<code>randomizedFilmList()</code>	Returns a list of all <code>FilmID</code> numbers in the ows database, in random order. Internally, this uses the <code>listRandomize()</code> method to perform the randomization.

Table 24.7 (CONTINUED)

METHOD	DESCRIPTION
<code>listRandomize(list)</code>	Accepts any comma-separated list and returns a new list with the same items in it, but in a random order. Because this method uses <code>access="Private"</code> , it can only be used internally within the <code>FilmRotationCFC</code> . This method really doesn't have anything to do with this CFC in particular; you could reuse it in any situation where you wanted to randomize a list of items.

TIP

In this CFC, I am adopting a convention of starting all public method names with the word `Get`. You might want to consider using naming conventions such as this when creating your own component methods.

It is conventional in many programming languages to start the name of any function that returns a Boolean value with the word `Is`. You might want to consider doing the same in your own CFCs.

Building FilmRotationCFC

Listing 24.4 shows the code for the `FilmRotationCFC` component. Because this component includes a number of methods, this code listing is a bit long. Don't worry. The code for each of the individual methods is quite short.

Listing 24.4 `FilmRotationCFC.cfc`—Building a CFC That Maintains Instance Data

```
<!--
Filename: FilmRotationCFC.cfc
Author: Nate Weiss (NMW)
Purpose: Creates FilmRotationCFC, a ColdFusion Component
-->

<cfcomponent output="false" hint="Provide Randomized Film List Functions">
  <cfproperty name="currentListPos" hint="current position in list" type="numeric"
required="no" default="1">
  <cfproperty name="filmList" hint="randomized list of films" type="string">
  <cfproperty name="rotationInterval" hint="how often the film rotates, in seconds"
type="numeric" required="no" default="5">
  <cfproperty name="currentUntil" hint="when does this film expire, and the next in
the list becomes current" type="date">

  <!-- ** begin initialization code ** -->
  <cfset THIS.filmList = randomizedFilmList()>
  <cfset THIS.currentListPos = 1>
  <cfset THIS.rotationInterval = 5>
  <cfset THIS.currentUntil = dateAdd("s", THIS.rotationInterval, now())>

  <!-- ** end initialization code ** -->
  <!-- Private function: RandomizedFilmList() -->
  <cffunction name="randomizedFilmList" returnType="string" access="private"
output="false"
hint="For internal use. Returns a list of all Film IDs, in random order.">
```

Listing 24.4 (CONTINUED)

```

<!-- This variable is for this function's use only -->
<cfset var getFilmIDs = "">

<!-- Retrieve list of current films from database -->
<cfquery name="getFilmIDs" datasource="ows"
cachedwithin="#CreateTimeSpan(0,1,0,0)#">
SELECT FilmID FROM Films
ORDER BY MovieTitle
</cfquery>

<!-- Return the list of films, in random order -->
<cfreturn listRandomize(valueList(getFilmIDs.FilmID))>
</cffunction>

<!-- Private utility function: ListRandomize() -->
<cffunction name="listRandomize" returnType="string"
output="false"
hint="Randomizes the order of the items in any comma-separated list.">

<!-- List argument -->
<cfargument name="list" type="string" required="Yes"
hint="The string that you want to randomize.">

<!-- These variables are for this function's use only -->
<cfset var result = "">
<cfset var randPos = "">

<!-- While there are items left in the original list... -->
<cfloop condition="listLen(ARGUMENTS.list) gt 0">
  <!-- Select a list position at random -->
  <cfset randPos = randRange(1, listLen(ARGUMENTS.list))>
  <!-- Add the item at the selected position to the Result list -->
  <cfset result = listAppend(result, listGetAt(ARGUMENTS.list, randPos))>
  <!-- Remove the item from selected position of the original list -->
  <cfset ARGUMENTS.list = listDeleteAt(ARGUMENTS.list, randPos)>
</cfloop>

<!-- Return the reordered list -->
<cfreturn result>
</cffunction>

<!-- Private method: IsFilmNeedingRotation() -->
<cffunction name="isFilmNeedingRotation" access="private" returnType="boolean"
output="false"
hint="For internal use. Returns TRUE if the film should be rotated now.">

  <!-- Compare the current time to the THIS.CurrentUntil time -->
  <!-- If the film is still current, DateCompare() will return 1 -->
  <cfset var dateComparison = dateCompare(THIS.currentUntil, now())>

  <!-- Return TRUE if the film is still current, FALSE otherwise -->
  <cfreturn dateComparison neq 1>
</cffunction>

<!-- RotateFilm() method -->
<cffunction name="rotateFilm" access="private" returnType="void" output="false"

```

Listing 24.4 (CONTINUED)

```

hint="For internal use. Advances the current movie.">

    <!-- If the film needs to be rotated at this time... -->
    <cfif isFilmNeedingRotation()>
        <!-- Advance the instance-level THIS.CurrentListPos value by one -->
        <cfset THIS.currentListPos = THIS.currentListPos + 1>

        <!-- If THIS.CurrentListPos is now more than the number of films, -->
        <!-- Start over again at the beginning (the first film) -->
        <cfif THIS.currentListPos gt listLen(THIS.FilmList)>
            <cfset THIS.currentListPos = 1>
        </cfif>

        <!-- Set the time that the next rotation will be due -->
        <cfset THIS.currentUntil = dateAdd("s", THIS.rotationInterval, now())>
    </cfif>
</cffunction>

<!-- Private method: CurrentFilmID() -->
<cffunction name="currentFilmID" access="private" returnType="numeric"
output="false"
hint="For internal use. Returns the ID of the current film in rotation.">

    <!-- Return the FilmID from the current row of the GetFilmIDs query -->
    <cfreturn listGetAt(THIS.filmList, THIS.currentListPos)>
</cffunction>

<!-- Public method: GetCurrentFilmID() -->
<cffunction name="getCurrentFilmID" access="public" returnType="numeric"
output="false"
hint="Returns the ID number of the currently 'featured' film.">
    <!-- First, rotate the current film -->
    <cfset rotateFilm()>

    <!-- Return the ID of the current film -->
    <cfreturn currentFilmID()>
</cffunction>

<!-- Public method: GetCurrentFilmData() -->
<cffunction name="getCurrentFilmData" access="remote" returnType="struct"
output="false"
hint="Returns structured data about the currently 'featured' film.">

    <!-- This variable is local just to this function -->
    <cfset var currentFilmData = "">

    <!-- Invoke the GetCurrentFilmID() method (in separate component) -->
    <!-- Returns a structure with film's title, summary, actors, etc. -->
    <cfinvoke component="FilmDataCFC2" method="getFilmData"
filmID="#getCurrentFilmID()#" returnVariable="currentFilmData">

        <!-- Return the structure -->
        <cfreturn currentFilmData>
    </cffunction>
</cfcomponent>

```

The most important thing to note and understand about this CFC is the purpose of the first few `<cfset>` tags at the top of Listing 24.4. Because these lines sit directly within the body of the `<cfcomponent>` tag, outside any `<cffunction>` blocks, they are considered *initialization code* that will be executed whenever a new instance of the component is created. Notice that each of these `<cfset>` tags creates variables in the special `THIS` scope, which means they are assigned to each instance of the component separately. Typically, all that happens in a CFC's initialization code is that it sets instance data in the `THIS` scope.

NOTE

It's important to understand that these lines don't execute each time one of the instance's methods is called. They execute only when a new instance of the component is brought to life with the `<cfobject>` tag.

The `<cfset>` tags at the top of the listing create these instance variables:

- `THIS.filmList` is a list of all current films, in the order in which the component should show them. The component's `randomizedFilmList()` method creates the sequence. This order will be different for each instance of the CFC.
- `THIS.currentListPos` is the current position in the randomized list of films. The initial value is 1, which means that the first film in the randomized list will be considered the featured film.
- `THIS.rotationInterval` is the number of seconds that a film should be considered featured before the component features the next film. Right now, the interval is 5 seconds.
- `THIS.currentUntil` is the time at which the current film should be considered expired. At that point, the CFC will select the next film in the randomized list of films. When the component is first instantiated, this variable is set to 5 seconds in the future.

Let's take a quick look at the `<cffunction>` blocks in Listing 24.4.

The `randomizedFilmList()` method will always be the first one to be called, since it is used in the initialization code block. This method simply retrieves a record set of film IDs from the database. Then it turns the film IDs into a comma-separated list with ColdFusion's `valueList()` function and passes the list to the CFC's `listRandomize()` method. The resulting list (which is a list of films in random order) is returned as the method's return value.

The `listRandomize()` method uses a combination of ColdFusion's list functions to randomize the list supplied to the `list` argument. The basic idea is to pluck items at random from the original list, adding them to the end of a new list called `result`. When there are no more items in the original list, the `result` variable is returned as the method's return value.

The `currentFilmID()` method simply returns the `FilmID` in the current position of the CFC's randomized list of films. As long as `THIS.currentListPos` is set to 1, this method returns the first film's ID.

The `isFilmNeedingRotation()` method uses `dateCompare()` to compare `THIS.currentUntil` to the current time. If the time has passed, this method returns `TRUE` to indicate that the current film is ready for rotation.

The `rotateFilm()` method is interesting because it actually makes changes to the variables in the `THIS` scope first created in the initialization code block. First, it uses `isFilmNeedingRotation()` to see whether the current film has been featured for more than 5 seconds already. If so, it advances the `This.currentListPos` value by 1. If the new `currentListPos` value is greater than the length of the list of films, that means all films in the sequence have been featured, so the position is set back to 1. Lastly, the method uses ColdFusion's `dateAdd()` function to set the `THIS.currentUntil` variable to 5 seconds in the future.

The `getCurrentFilmID()` method ties all the concepts together. Whenever this method is used, the `rotateFilm()` method is called (which will advance the current film to the next item in the sequence if the current one has expired). It then calls `currentFilmID()` to return the current film's ID.

Storing CFCs in the APPLICATION Scope

Now that the `FilmRotationCFC` component is in place, it's quite simple to put it to use. Listing 24.5 shows one way of using the component.

Listing 24.5 `UsingFilmRotationCFCa.cfm`—Instantiating a CFC at the Application Level

```
<!--
  Filename: UsingFilmRotationCFCa.cfm
  Author: Nate Weiss (NMW)
  Purpose: Demonstrates storage of CFC instances in shared memory scopes
-->

<html>
<head>
  <title>Using FilmRotationCFC</title>
</head>

<body>

<!-- If an instance of the FilmRotatorCFC component hasn't been created --->
<!-- yet, create a fresh instance and store it in the APPLICATION scope --->
<cfif not isDefined("APPLICATION.filmRotator")>
  <cfset APPLICATION.filmRotator = new FilmRotationCFC()>
</cfif>

<!-- Invoke the GetCurrentFilmID() method of the FilmRotator CFC object --->
<cfset featuredFilmID = Application.filmRotator.getCurrentFilmID()>

<p>The callout at the right side of this page shows the currently featured film.
The featured film changes every five seconds.
Just reload the page to see the next film in the sequence.
The sequence will not change until the ColdFusion server is restarted.</p>

<!-- Show the current film in a callout, via custom tag --->
<cf_ShowMovieCallout
  filmID="#featuredFilmID#">

</body>
</html>
```

The idea here is to keep an instance of `FilmRotationCFC` in the `APPLICATION.filmRotator` variable. Keeping it in the `APPLICATION` scope means that the same instance will be kept in the server's memory until the ColdFusion server is restarted. All sessions that visit the page will share the instance.

First, a simple `isDefined()` test sees if the CFC instance called `APPLICATION.filmRotator` already exists. If not, the instance is created with the `<cfobject>` tag. So, after this `<cfif>` block, the instance is guaranteed to exist. Keep in mind that the CFC's initialization code block is executed when the instance is first created.

NOTE

If you wanted the CFC instance to be available to all pages in the application, you could move the `<cfif>` block in Listing 24.5 to your `Application.cfc` file.

Displaying the currently featured film is simply a matter of calling the `getCurrentFilmID()` method and passing it to the `<cf_ShowMovieCallout>` custom tag. When a browser visits this listing, the currently featured movie is displayed. If you reload the page repeatedly, you will see that the featured movie changes every 5 seconds. If you wait long enough, you will see the sequence of films repeat itself. The sequence will continue to repeat until the ColdFusion server is restarted, at which point a new sequence of films will be selected at random.

Storing CFCs in the SESSION Scope

One of the neat things about CFCs is their independence. You will note that the code for the `RotateFilmCFC` component doesn't contain a single reference to the `APPLICATION` scope. In fact, it doesn't refer to any of ColdFusion's built-in scopes at all, except for the `THIS` scope.

This means it's possible to create some instances of the CFC that are kept in the `APPLICATION` scope, and others that are kept in the `SESSION` scope. All the instances will work properly and will maintain their own versions of the variables in the `THIS` scope.

To see this in action, go back to Listing 24.5 and change the code so that the CFC instance is kept in the `SESSION` scope instead of the `APPLICATION` scope. Now each Web session will be given its own `FilmRotator` object, stored as a session variable. You can see how this looks in Listing 24.6 (in the upcoming section "Modifying Properties from a ColdFusion Page").

To see the difference in behavior, open the revised listing in two different browsers (say, Firefox and Internet Explorer 8), and experiment with reloading the page. You will find that the films are featured on independent cycles and that each session sees the films in a different order. If you view the page on different computers, you will see that each machine also has its own private, randomized sequence of featured films.

Instance Data as Properties

As I've explained, the code for the `FilmRotationCFC` component uses the `THIS` scope to store certain variables for its own use. You can think of these variables as *properties* of each component instance, because they are the items that make a particular instance special, giving it its individuality, its life.

Sometimes you will want to display or change the value of one of these properties from a normal ColdFusion page. ColdFusion makes it very easy to access an instance's properties. Basically, you can access any variable in a CFC's `THIS` scope as a property of the instance itself.

Modifying Properties from a ColdFusion Page

If you have a CFC instance called `SESSION.myFilmRotator` and you want to display the current value of the `currentUntil` property (that is, the value of the variable that is called `THIS.currentUntil` within the CFC code), you can do so with the following in a normal `.cfm` page:

```
<cfoutput>
  #timeFormat(SESSION.myFilmRotator.currentUntil)#
</cfoutput>
```

To change the value of the `rotationInterval` property (referred to as `THIS.rotationInterval` in the `FilmRotationCFC.cfc` file) to 10 seconds instead of the usual 5 seconds, you could use this line:

```
<cfset SESSION.myFilmRotator.rotationInterval = 10>
```

After you changed the `rotationInterval` for the `SESSION.FilmRotator` instance, then that session's films would rotate every 10 seconds instead of every 5 seconds. Listing 24.6 shows how all this would look in a ColdFusion page.

Listing 24.6 UsingFilmRotationCFCb.cfm—Interacting with a CFC's Properties

```
<!---
  Filename: UsingFilmRotationCFCc.cfm
  Author: Nate Weiss (NMW)
  Purpose: Demonstrates storage of CFC instances in shared memory scopes
  --->

<html>
<head>
  <title>Using FilmRotationCFC</title>
</head>

<body>

<!--- If an instance of the FilmRotatorCFC component hasn't been created --->
<!--- yet, create a fresh instance and store it in the SESSION scope --->
<cfif not isDefined("SESSION.myFilmRotator")>
  <cfset SESSION.myFilmRotator = new FilmRotationCFC()

  <!--- Rotate films every ten seconds --->
  <cfset SESSION.myFilmRotator.rotationInterval = 10>
</cfif>

<!--- Display message --->
<cfoutput>
  <p>
    The callout at the right side of this page shows the currently featured film.
    Featured films rotate every #SESSION.myFilmRotator.rotationInterval# seconds.
    Just reload the page to see the next film in the sequence.
    The sequence will not change until the web session ends.</p>
    The next film rotation will occur at:
```

Listing 24.6 (CONTINUED)

```
#timeFormat(SESSION.myFilmRotator.currentUntil, "h:mm:ss tt")#
</cfoutput>

<!-- Show the current film in a callout, via custom tag -->
<cf_ShowMovieCallout filmID="#SESSION.myFilmRotator.getCurrentFilmID()#">

</body>
</html>
```

NOTE

You can experiment with changing the `RotationInterval` property to different values. Keep in mind that the code in the `<cfif>` block will execute only once per session, so you may need to restart ColdFusion to see a change. If you are using J2EE Session Variables, you can just close and reopen your browser. Or you could move the `<cfset>` line outside the `<cfif>` block.

What all this means is that the CFC's methods can access an instantiated CFC's properties internally via the `THIS` scope, and your ColdFusion pages can access them via the instance object variable itself. As you learned in the introduction to this topic, CFCs can be thought of as containers for data and functionality, like many objects in the real world. You know how to access the data (properties) as well as the functionality (methods).

Documenting Properties with `<cfproperty>`

As you learned earlier, you can easily view a CFC's methods in the Component tree in the Dreamweaver's Application panel. You can also view them in the automatically generated reference page that ColdFusion produces if you visit a CFC's URL with your browser. Since a CFC's properties are also important, it would be nice if there was an easy way to view them too.

ColdFusion provides a tag called `<cfproperty>` that lets you provide information about each variable in the `this` scope that you want to document as an official property of a component. The `<cfproperty>` tags must be placed at the top of the CFC file, just within the `<cfcomponent>` tag, before any initialization code.

Another function that `<cfproperty>` provides for you is the ability to do type checking on your properties (this is new in ColdFusion 9). Using the attributes `validate` and `validateparams`, you can specify the data types that are allowed for your components' properties, and if something tries to set them to an invalid value, ColdFusion will throw an error.

Table 24.8 shows the syntax for the `<cfproperty>` tag.

Table 24.8 `<cfproperty>` Tag Syntax

ATTRIBUTE	DESCRIPTION
<code>name</code>	The name of the property. This should match the name of the variable in the <code>THIS</code> scope that is used within the component's methods.
<code>type</code>	The data type of the property, such as <code>numeric</code> , <code>string</code> , or <code>query</code> .
<code>required</code>	Whether the property is required (documentation only).

Table 24.8 (CONTINUED)

ATTRIBUTE	DESCRIPTION
default	The initial value of the property (documentation only).
hint	An explanation of what the property does or represents.
displayName	An alternate name for the property.
validate	Data type for the parameter. See the “Implicit Getters and Setters” section below.
validateparams	Parameters required for the validation type, such as if, range, max, and so on. See the “Implicit Getters and Setters” section below.

You’ll notice in Listing 24.6 that I’ve documented all the properties with `<cfproperty>`, for example:

```
<cfproperty
  name="RotationInterval"
  type="numeric"
  required="No"
  default="5"
  hint="The number of seconds between film rotations.">
```

NOTE

Remember that `<cfproperty>` doesn’t actively create a property in this version of ColdFusion. Just because you add the `<cfproperty>` tag to document the `THIS.rotationInterval` property doesn’t mean that you can remove the `<cfset>` tag that actually creates the variable and gives it its initial value.

CFCs and the VARIABLES Scope

The `THIS` scope isn’t the only way to persist data within a CFC. Each CFC also has a `VARIABLES` scope. This scope acts just like the `VARIABLES` scope within a simple CFM page. Like the `THIS` scope, each method in the CFC can read and write to the scope. However, unlike the `THIS` scope, you can’t display or modify the value outside the CFC.

Some people consider this a good thing. Look at the code in Listing 24.6. One line sets the CFC’s `rotationInterval` variable. What happens if the code sets it to a value of “ten” instead of the number “10”? The next time this page, or any other, runs the `getCurrentFilmID` method, the code will blow up because the property is no longer a number. The whole point of encapsulation is to prevent problems like this. How can you prevent this?

Keeping Your Properties Clean: Getters and Setters

To make sure your properties are the right type, in the right range, and so on, you generally don’t want to allow your users to directly access the properties of your CFCs. To prevent this, for each property of your CFC, build two methods—one to get the current value and another to set its value. This allows you to check and make sure the value being set is valid, as well as massage any data on the way out so code that’s getting a property’s value gets it in a way that’s useful. For example, instead of directly accessing the `rotationInterval` value of the CFC, the CFC itself could define a

setRotationInterval method. Any CFM that needs to set this value would simply use the method. If an invalid value is passed in, the component can throw an error or simply ignore it.

It's considered good programming practice to always name the getters and setters getProperty() and setProperty(). For example, for the currentUntil property, you would name them getCurrentUntil() and setCurrentUntil().

Listing 24.7 shows an excerpt from the FilmRotationCFC that contains a typical getter and setter for the rotationInterval property.

Listing 24.7 FilmRotationCFCb.cfc—Film Rotation with Getters and Setters (Excerpt)

```
<!-- getter method for rotationInterval --->
<cffunction name="getrotationInterval" returnType="numeric" hint="getter for
rotationInterval property">
  <cfreturn this.currentUntil>
</cffunction>

<!-- setter method for rotationInterval --->
<cffunction name="setrotationInterval" returnType="void" hint="setter for
rotationInterval property">
  <cfargument name="newValue" required="yes" type="numeric" hint="new value for
rotationInterval property">
  <cfif isNumeric(arguments.newValue)>
    <cfset this.currentUntil = arguments.newValue>
  <cfelse>
    <cfthrow type="application" message="Invalid value for setrotationInterval: must
be numeric">
  </cfif>
  <cfreturn>
</cffunction>
```

Implicit Getters and Setters

Since all getter and setter methods generally do is validate the data being set, most of these methods wind up looking almost exactly the same. Because of this, the folks who designed the ColdFusion 9 language added a feature to components that lets ColdFusion provide this functionality without you actually having to write the functions.

All you need to do to add getter and setter functions to all your properties is add a single attribute to the <cfcomponent> tag: `accessors="true"`. For example, in `FilmRotationCFC.cfc`, you'd simply change the first line to read as follows:

```
<cfcomponent output="false" hint="Provide Randomized Film List Functions"
  accessors="true">
```

Without any additional work, you can now call getters and setters for any of the properties you have defined with <cfproperty> tags. `getFilmList`, `setFilmList`, `getCurrentListPos`, `setCurrentListPos`, and so on, are all now available.

In addition, you can now use the `validate` and `validateparams` attributes of `<cfproperty>` to automatically create the validation code for your properties. To effectively create the `setrotationInterval` validation in Listing 24.7, you'd just change your `<cfproperty>` tag to this:

```
<cfproperty name="rotationInterval" hint="how often the film rotates, in seconds"
type="numeric" validate="numeric">
```

NOTE

If you explicitly define getters and setters for some of your properties, ColdFusion won't override them if you have `accessors=true`. Also, if you don't want certain properties to have a getter or setter, for those individual properties you can add a `"getter=no"` or `"setter=no"` attribute to the `<cfproperty>` tag, and ColdFusion won't create them.

Initializing Components

Most of the time, when you create a component, you'll want it to be independent of the application that is calling it. For example, you wouldn't want to hard-code the value of the data source into a component, because it would be different between one application using it and another. However, the component isn't usable until it "knows" what its data source is and probably some other initialization values. Therefore, most components require some sort of initialization to work.

The typical thing to do with a component is create a special method called `init` that is called when starting up the component; it returns a reference to the component itself. ColdFusion 9 supports this behavior with the `new` keyword by automatically calling the `init` method with whatever arguments are passed to the component when it is invoked.

Listing 24.8 contains an excerpt of the `init` method from the updated version of the `FilmRotationCFC` that contains the `init` function. Note that the code outside the methods has been moved to the `init` method, and the `<cfquery>` tag in the `randomizedFilmList` method now uses the local variable `variables.dsn` instead of the hard-coded value for the data source.

Listing 24.8 Updated `FilmRotationCFC.cfc` with `init` Method

```
<!---
Filename: FilmRotationCFC.cfc
Author: Nate Weiss (NMW)
Purpose: Creates FilmRotationCFC, a ColdFusion Component
-->
<cfcomponent output="false">
  <cfproperty name="currentListPos" hint="current position in list" type="numeric">
  <cfproperty name="filmList" hint="randomized list of films" type="string">
  <cfproperty name="rotationInterval" hint="how often the film rotates, in seconds"
type="numeric">
  <cfproperty name="currentUntil" hint="when does this film expire, and the next in
the list becomes current" type="date">
  <!--- *** begin initialization code *** -->
  <!--- init method -->
  <cffunction name="init" returnType="component" hint="initialization">
    <cfargument name="datasource" required="yes" type="string">
    <cfargument name="rotationInterval" required="no" default="5" type="numeric">
    <cfset variables.dsn = arguments.datasource>
    <cfset THIS.rotationInterval = arguments.rotationInterval>
    <cfset THIS.filmList = randomizedFilmList(>
```

Listing 24.8 (CONTINUED)

```

    <cfset THIS.currentListPos = 1>
    <cfset THIS.currentUntil = dateAdd("s", THIS.rotationInterval, now())>
</cffunction>
<!-- *** end initialization code *** -->
<!-- Private function: RandomizedFilmList() -->
<cffunction name="randomizedFilmList" returnType="string" access="private"
output="false"
hint="For internal use. Returns a list of all Film IDs, in random order.">

    <!-- This variable is for this function's use only -->
    <cfset var getFilmIDs = "">

    <!-- Retrieve list of current films from database -->
    <cfquery name="getFilmIDs" datasource="#variables.dsn#"
cachedwithin="#CreateTimeSpan(0,1,0,0)#">
    SELECT FilmID FROM Films
    ORDER BY MovieTitle
    </cfquery>

    <!-- Return the list of films, in random order -->
    <cfreturn listRandomize(valueList(getFilmIDs.FilmID))>
</cffunction>

```

To call this new method, you change the line in Listing 24.5 that creates the component variable to pass in the required arguments, as shown here:

```

<cfset APPLICATION.FilmRotator.c = new FilmRotationCFCc(datasource="ows",
rotationInterval="#variables.rotInterval#")>

```

NOTE

It's standard practice to use the initialization method `init`, but ColdFusion allows you to override that value by passing the argument `initmethod=<methodname>`.

CFCs, Shared Scopes, and Locking

In Chapter 18, “Introducing the Web Application Framework,” in Volume 1, you learned that it’s important to beware of *race conditions*. A race condition is any type of situation where strange, inconsistent behavior might arise if multiple page requests try to change the values of the same variables at the same time. Race conditions aren’t specific to ColdFusion development; all Web developers should bear them in mind. See Chapter 18 for more information about this important topic.

Since the past few examples have encouraged you to consider storing instances of your CFCs in the `APPLICATION` or `SESSION` scope, you may be wondering whether there is the possibility of logical race conditions occurring in your code and whether you should use the `<cflock>` tag or some other means to protect against them if necessary.

The basic answer is that packaging your code in a CFC doesn’t make it more or less susceptible to race conditions. If the nature of the information you are accessing within a CFC’s methods is such that it shouldn’t be altered or accessed by two different page requests at the same time, you

most likely should use the `<cflock>` tag to make sure one page request waits for the other before continuing.

Direct Access to Shared Scopes from CFC Methods

If your CFC code is creating or accessing variables in the `APPLICATION` or `SESSION` scope directly (that is, if the words `APPLICATION` or `SESSION` appear in the body of your CFC's `<cffunction>` blocks), place `<cflock>` tags around those portions of the code. The `<cflock>` tags should appear inside the `<cffunction>` blocks, not around them. Additionally, you should probably place `<cflock>` tags around any initialization code (that is, within `<cfcomponent>` but outside any `<cffunction>` blocks) that refers to `APPLICATION` or `SESSION`. In either case, you would probably use `scope="SESSION"` or `scope="APPLICATION"` as appropriate; alternatively, you could use `<cflock>`'s `NAME` attribute as explained in Chapter 18 if you wanted finer-grained control over your locks.

Also, ask yourself why you're even using the `APPLICATION` or `SESSION` scope in your CFC code. Is it really necessary? If the idea is to persist information, why not simply store the CFC itself in one of the persistent scopes? This will be helpful if you decide that the information needs to be specific to the `SESSION` and not to the `APPLICATION`. If you never directly referenced any scope in your CFC code but instead simply stored the CFC in one of the scopes, "moving" the CFC then becomes a simple matter.

Locking Access to the THIS Scope

The `FilmRotationCFC` example in this chapter (Listing 24.4) doesn't manipulate variables in the `APPLICATION` or `SESSION` scope; instead, the CFC is designed so that entire instances of the CFC can be stored in the `APPLICATION` or `SESSION` scope (or the `SERVER` scope, for that matter) as the application's needs change over time. This is accomplished by only using variables in the `THIS` scope, rather than referring directly to `SESSION` or `APPLICATION`, within the CFC's methods.

You may wonder how to approach locking in such a situation. I recommend that you create a unique lock name for each component when each instance is first instantiated. You can easily accomplish this with ColdFusion's `CreateUUID()` function. For instance, you could use a line like this in the component's initialization code, within the body of the `<cfcomponent>` tag:

```
<cfset THIS.lockName = CreateUUID()>
```

The `THIS.lockName` variable (or property, if you prefer) is now guaranteed to be unique for each instance of the CFC, regardless of whether the component is stored in the `APPLICATION` or `SERVER` scope. You can use this value as the name of a `<cflock>` tag within any of the CFC's methods. For instance, if you were working with a CFC called `ShoppingCartCFC` and creating a new method called `addItemToCart()`, you could structure it according to this basic outline:

```
<cffunction name="addItemToCart">
  <cflock name="#THIS.lockName#" type="Exclusive" timeout="10">
  <!-- Changes to sensitive data in THIS scope goes here -->
  </cflock>
</cffunction>
```

For more information on the `<cflock>` tag, especially when to use `type="Exclusive"` or `type="ReadOnly"`, see the “Using Locks to Protect Against Race Conditions” section in Chapter 18.

Working with Inheritance

Frequently it is useful to have components that implement similar functionality in different ways. For example, you might have a `circle` component and a `square` component. Each has a `draw` method: The `circle` component draws a circle on the screen, and the `square` component draws a square. Each also has independent properties. For example, the `circle` has `circumference`, and the `square` has `length`. The two components also have a lot in common: They each have `perimeter` and `area`. The `square` and `circle` components are special cases of a shape; they have everything a shape has, plus more. Thus, it would make sense to create a single parent component called `shape` that has the information that is common to all types of shapes and then to have `child` components that inherit this information and also add their own. Thus, `square`, as a child of `shape`, would have all the things that `shape` has plus `length`, and it would implement its own variation of `draw`.

Just as you can use the word *my* to refer to a CFC’s `THIS` scope (“*my* ID is 123 and *my* first name is Fred...”), in *inheritance*, you can think of the words *is a*. A square *is a* shape. An actor *is a* person. A cat *is a* mammal. In these cases, actor, square, and cat are children of person, shape, and mammal. Some parents can exist by themselves; there can be a person who is not an actor. Some other parents, though, are abstract; shape can’t draw itself without knowing what shape it is. Rather, the parent is intended as more of a handy template upon which more specific things can be based.

In a movie studio application, actors and directors are both types of people, with some properties that are common and some that are unique. So, for types of people, you could create a component to represent a person and have each of these variants inherit from it.

Listing 24.9 shows the basic person component. It has a first name and last name (stored in the `THIS` scope) and has one function that “shows” the person by outputting the first and last names.

Listing 24.9 person.cfc—The Basic person Component

```
<!--
  Filename: Person.cfc
  Author: Ken Fricklas (KF)
  Purpose: Basic Person CFC
-->
<cfcomponent hint="Parent Component - Person">

  <cfparam name="THIS.firstName" default="John">
  <cfparam name="THIS.lastName" default="Doe">

  <cffunction name="showPerson" output="true" hint="showPerson in person.cfc" >
    <B>#THIS.firstName# #THIS.lastName#</B>
  </cffunction>

</cfcomponent>
```

A component inherits from a parent component with the `EXTENDS` attribute of the `<cfcomponent>` tag. The value of the attribute is the name of the component upon which the new component should be based. Thus, a director component could consist of nothing more than Listing 24.10.

Listing 24.10 `director.cfc`—The director Component

```
<!---
  Filename: Director.cfc
  Author: Ken Fricklas (KF)
  Purpose: A Minimal Inherited CFC
  --->
<cfcomponent displayName="Movie Director" extends="person">
</cfcomponent>
```

Now, the director is an exact copy of the person component and has inherited all the properties and methods of its parent. A CFML page, then, could create an instance of the director and invoke the methods of the person component as though they were part of the director component (Listing 24.11).

Listing 24.11 `showDirector.cfm`—Display the Director

```
<!---
  Filename: showDirector.cfm
  Author: Ken Fricklas (KF)
  Purpose: Show the director
  --->
<cfset cfcDirector = new Director()>
<cfoutput>#cfcDirector.showPerson()#</cfoutput>
```

NOTE

In fact, every component inherits from a root component called `WEB-INF.cftags.component`. This component is the mother of all components. In its default case, it is simply an empty file without any functions or properties, but you can implement custom logging, debugging, and other behavior by modifying this root component.

Overriding Properties and Methods

Just because the parent does something doesn't mean that the child is stuck with it. The component can override parts of the parent component. If you want the director component to set the `firstName` and `lastName` properties to different values than those of the person component, you simply add code that redefines the values. The director, because it's the one being invoked, will take precedence. So, the director component is now coded like this:

```
<cfcomponent displayName="Movie Director" extends="person">
  <cfset THIS.firstName = "Jim">
  <cfset THIS.lastName = "Jarofmush">
</cfcomponent>
```

When invoked from the CFML page, this component now will output “Jim Jarofmush” instead of “John Doe.” The `THIS` scope assignments made in a child override those of its parent. Likewise,

adding a `showPerson` function to the `director` component will override the `showPerson` function from the parent:

```
<cffunction name="showPerson" output="true" hint="showPerson in director.cfc">
  <B>A swell director named #THIS.firstName# #THIS.lastName#</B>
</cffunction>
```

Using the SUPER Scope

What if a child component needs to use the functionality in a method in its parent, but it has redefined that method already? In the `director` component, you could call the parent `showPerson` method, but you want to add your own information to it. You do this with the special scope `SUPER`. `SUPER` acts similarly to `THIS`, but instead of referring to a property or method in the current component, it refers to the property or method in the component's parent. You could redefine `showPerson` in the `director` component as shown in Listing 24.12.

Listing 24.12 Final `director.cfc`

```
<!--
  Filename: director.cfc
  Author: Ken Fricklas (KF)
  Purpose: Demonstrates use of super and property overrides
-->
<cfcomponent displayName="Movie Director" extends="person">
  <cfset THIS.firstName = "Jim">
  <cfset THIS.lastName = "Jarofmush">
  <cffunction name="showPerson" output="true" hint="showPerson in director.cfc">
    <B>A swell director - #super.showPerson()#</B>
  </cffunction>
</cfcomponent>
```

This code calls the `showPerson` method in the `person` component.

NOTE

In addition to the child being able to invoke functions that are really part of the parent component (and overriding them, if desired), the parent can call functions that are part of the child by referencing them in the instance's `THIS` scope. This technique can be useful when multiple components are descendants of the same parent but require slightly different methods.

Must you use inheritance in your ColdFusion applications? Certainly not. But it can be very useful in ways similar to other code-reuse techniques. Components can be built-in “branches,” as in a family tree with chains of ancestral parent components, each providing base functionality to their children.

Component packages can help with this type of organization, too, to make applications more easily maintainable. In that case, the `extends="..."` attribute uses the same package path syntax as a `<cfinclude>` tag. For example, to inherit from a component called `person` in the package `myApp.components`, the `<cfcomponent>` tag would be coded like this:

```
<cfcomponent extends="myApp.components.person">
```

NOTE

Inheritance can also be more than one level deep. Just as `actor` is a special case of `person`, `comedian` is a special case of `actor`. `comedian` could extend `actor`, which extends `person`. Then `comedian` would inherit methods and properties from `actor`, which inherits from `person`. `Super()` can also be chained; `super.super.showPerson()` is a valid construct, if you wanted to run the `showPerson()` from `person` directly from `comedian` and bypass an override method in `actor`.

Defining Interfaces

When designing components, it's frequently useful to define a template for anyone passing components to general functions. For example, you might create a function that takes as an argument an instance of a cast or crew member of a movie, calls a method named `getResume` to get a copy of the crew member's résumé as a query, and calls another method named `showPersonHTML` to show the crew member's name and information in an HTML display.

When requesting several implementations of this function to implement different types of cast members, actors, directors, producers, and so on, you might want these components to each do everything differently—you don't want them to inherit from a common parent, but they all have to implement a minimum set of functionality to meet the requirements of the system.

The definition of functions without actual implementation is a special type of component definition called an *interface*. An interface is basically a contract between a component and a system. You define an interface in a file with an extension of `.cfc`, but instead of enclosing the component with `<cfcomponent?>` tags, you use a tag called `<cfinterface>`. Any component that implements the interface must contain all the methods defined in it, or the system will display an error.

Listing 24.13 contains an interface for the component just described.

Listing 24.13 `iCastCrew.cfc`—An Interface

```
<cfinterface hint="cast or crewmember interface">
  <cffunction name="getResume" access="public" returnType="query" hint="return
resume as query">
  </cffunction>
  <cffunction name="showPersonHTML" access="public" returnType="string" hint="show
person information as HTML">
  <cfargument name="detail" type="boolean" required="no" default="true"
hint="show detailed information">
  </cfargument>
</cffunction>
</cfinterface>
```

Introspection and inheritance (via the `extends` attribute of `<cfinterface>`) can be used with interfaces the same way as with components.

NOTE

Interfaces are typically given names that start with a lowercase *i*—for example, `iComponent.cfc`—to distinguish them from other components.

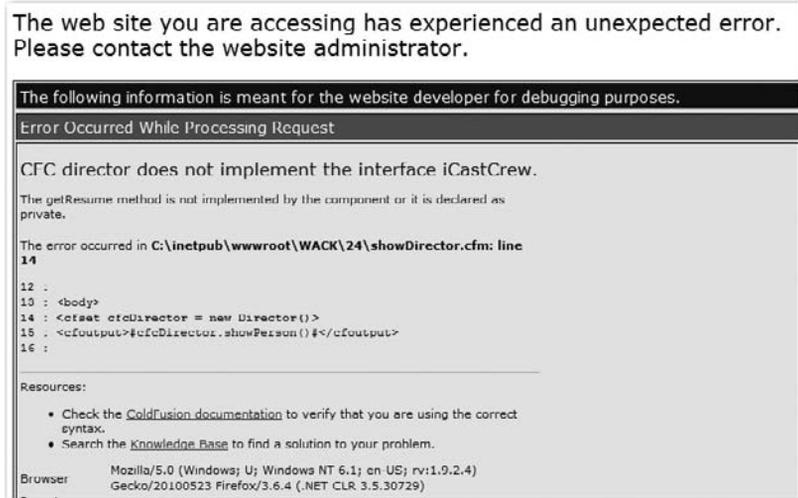
To make sure that a component implements an interface, you use the `implements` keyword in the `<cfcomponent>` tag; for example, to make the `director` an implementation of `iCastCrew`, you would change the first line to this:

```
<cfcomponent displayName="Movie Director" extends="person" implements="iCastCrew">
```

If you ran this code, you would get the error shown in Figure 24.7, since the `director` component is missing some of the functions defined in the interface.

Figure 24.7

Error results from a failed implementation of an interface.



Implementing Security

ColdFusion provides two ways to secure the functionality that you encapsulate in a ColdFusion component: roles-based authorization and access control. Chapter 52, “Understanding Security,” in Volume 3, discusses application user authentication and authorization, which allows the assignment of roles to your application’s users. This roles-based security can also be applied to the functions in a CFC. The second technique, access control, was used in the preceding chapter in every `<cffunction>` tag as the attribute `access="..."`.

Implementing Access Control

The `access` attribute of the `<cffunction>` tag basically answers the question, “Who can use this function?” The attribute has four options: `private`, `package`, `public` (the default), and `remote`. These four options represent, in that order, the degree of openness of the function.

The `access` options range from a narrow group of potential consumers to a very broad audience. The consumers allowed by each option are as follows:

- **Private.** Only other functions within the same CFC can invoke the function. This is great to hide the details of how your component is implemented and to keep sneaky

developers from writing code based on parts of your component that may not stay the same in future revisions. Note that private functions are inherited like any other methods.

- **Package.** Only components in the same package can invoke the function. This is just like `private`, except if you have implemented a system made up of multiple related CFCs, they can all use the methods declared `package`.
- **Public.** Any CFML template or CFC on the same server can invoke the function. This is the default.
- **Remote.** Any CFML template or CFC on the same server can invoke the function, as can programs running through the Web server. For example, Flash, Web Services, and methods can be invoked directly from a Web browser, as shown in the examples earlier in the chapter.

Implementing Role-Based Security in CFCs

In some applications, you'll want to control access to a component's functions based on who is using your application. This will be most common in traditional, HTML-based user interface applications, but it may also be true of Adobe Flash applications. Role-based security is not, however, a common approach to securing access to Web Services, since a Web Services client is a *program* and not an *individual*.

To see this technique in action, let's go back to the `director` component that was created earlier in this chapter—the one that retrieves information about all actors. Part of the Orange Whip Studios Web application allows studio executives to review the salaries of the stars—how much should the studio expect to fork over for their next box-office smash? Of course, this information is not exactly something that they want just anybody seeing.

First you need to create the basic security framework for this part of the application, with the security tags in ColdFusion: `<cflogin>`, `<cfloginuser>`, and `<cflogout>`. (I discuss this process in detail in Chapter 21, “Securing Your Applications,” in Volume 1.)

For the purposes of this exercise, you can test by running the `<cfloginuser>` tag with the role you want to test with:

```
<cfloginuser name="Test" password="dummy" roles="Producers">
```

Any roles for the logged-in user will be the roles that correspond to those listed in the component function—more on this after you create the function in Listing 24.14.

The function will be simple: It takes an actor ID as an argument, queries that actor's salary history, and returns a record set. Notice, though, that the `roles` attribute in the `<cffunction>` tag has a comma-delimited list of values. Only users who have authenticated and been assigned one or more of those roles will be allowed to invoke the method.

Listing 24.14 actor.cfc—The Salary Method

```

<!--
  Filename: actor.cfc
  Author: Ken Fricklas (KF)
  Purpose: Demonstrates roles
  -->
<cfcomponent name="actor" extends="person">
<cffunction name="init" returnType="component">
  <cfargument name="datasource" required="yes" type="string">
  <cfset variables.dsn = arguments.datasource>
  <cfreturn this>
</cffunction>
<cffunction name="getActorSalary" returnType="query" roles="Producers, Executives">
  <cfargument name="actorID" type="numeric" required="true"
    displayName="Actor ID" hint="The ID of the Actor">
  <cfquery name="salaries" dataSource="#variables.dsn#">
    SELECT Actors.ActorID, Actors.NameFirst, Actors.NameLast,
      FilmsActors.Salary, Films.MovieTitle
    FROM Films
    INNER JOIN (Actors INNER JOIN FilmsActors
      ON Actors.ActorID = FilmsActors.ActorID)
      ON Films.FilmID = FilmsActors.FilmID
    WHERE Actors.ActorID = #Arguments.actorID#
  </cfquery>
  <cfreturn salaries>
</cffunction>
</cfcomponent>

```

The roles assigned to this function are producers and executives—they don't want any prying eyes finding this sensitive data. All you need now, then, is a page to invoke the component—something simple, as in Listing 24.15.

Listing 24.15 showSalary.cfm—Show Salary Page

```

<!--
  Filename: showSalary.cfm
  Author: Ken Fricklas (KF)
  Purpose: Demonstrate CFC roles
  -->
<!-- Make sure they are logged in. Change roles to "User" to see what happens if
they don't have sufficient access. -->
<cfloginuser name="Test" password="dummy" roles="Producers">
<!-- Invoke actors component. getActorSalary method will fail unless
they have sufficient access. -->
<cfset cfcActor = new actor(datasource="ows")>
<cfset salaryHistory = cfcActor.getActorSalary(17)>
<h1>Salaries of our stars...</h1>
<cfoutput>
<H2>
#salaryHistory.NameFirst# #salaryHistory.NameLast#</H2>
<cfloop query="salaryHistory">
  #MovieTitle# - #dollarFormat(Salary)#<BR>
</cfloop>
</cfoutput>

```

ColdFusion now has all it needs to control the access to the component. When the `salaryHistory` method is invoked, since there are values specified in the `roles` attribute, a comparison is automatically made between the values in the `roles` attribute of the `<cffunction>` tag and those in the `roles` attributes that were set in the `<cfloginuser>` tag. If the user is not logged in, this function will fail.

A match will allow the function to be executed as usual; a failure will cause ColdFusion to throw the error “The Current user is not authorized to invoke this method.”

NOTE

As noted, an unauthorized attempt to execute a secured function causes ColdFusion to throw an error. Consequently, you should put a `<cftry>` around any code that invokes secured functions.

This is not the only way to secure component functionality, of course. You could use the `isUserInRole()` function to check a user’s group permissions before even invoking the function, or you could use Web server security for securing the CFML files themselves. The role-based security in CFCs is, however, a good option, particularly if you are already using the ColdFusion authentication/authorization framework in an application.

Using the OnMissingMethod Method

It would be nice if all the CFCs that we write could handle their own errors. Starting in ColdFusion 8, any component can have a special method named `onMissingMethod` that will run whenever code attempts to run a method that hasn’t been defined in it. You can use this method to serve several purposes:

- Implement custom error handling. The `onMissingMethod` method can be especially useful when methods in different child classes might be called, even though they are not implemented in a particular component. For example, you could use `onMissingMethod` to handle a call to `getActorSalary` made to a `director` component.
- Run different code for several methods in a common place. If the same code can take the place of several methods, `onMissingMethod` can provide a way to run the common code from a single point. This approach is not recommended, however; it’s more straightforward to define all the methods separately and call the common code from each.
- Act as a proxy that calls another object or component that will actually implement the function. For example, you could create a component that is empty except for `onMissingMethod` that takes any method passed to it and calls a Web Service on another machine that consumes the method name and its arguments and returns a value. This approach is a good way to implement a flexible, distributed system.

The `onMissingMethod` function takes exactly two arguments, which contain the name of the method that was being called and a structure with the arguments that were passed to it. For example, here is a simple `onMissingMethod` method:

```
<cffunction name="onMissingMethod">
    <cfargument name="missingMethodName" type="string">
```

```

    <cfargument name="missingMethodNameArguments" type="struct">
    Hey! You called <cfoutput>#arguments.missingMethodName#</cfoutput> and I haven't
    got one!
</cffunction>

```

NOTE

Since `onMissingMethod` always returns successfully, if you can't handle an error in this method, you should throw a new error.

Distributed CFCs and Serialization

I've already discussed storing CFCs in the session scope. One problem with storing CFCs in this way is that when more than one server is in use, each server has its own session variables; if a user moves to another server during the course of a visit to your site, the session data can be lost. One way to solve this problem is to create “sticky” sessions, which requires special software or hardware—but what if the Web server goes down?

Many Java application servers that ColdFusion runs on can support distributed sessions. Basically, what the servers do is “serialize” the data in the session scope, which means that a server writes the data in a flat form (turns it into a string, in the same way that CFWDDX does) and passes it to all other servers in the cluster that might process the new request. No matter which server is used, it has a copy of the session data.

Before ColdFusion 8, components could not be serialized, so they would not be passed between machines as part of the session. As of ColdFusion 8, however, components can be serialized and distributed.

NOTE

In Java parlance, this means that ColdFusion now supports the Java serializable interface. You can read more about this at <http://java.sun.com/developer/technicalArticles/Programming/serialization/>.

In addition, you can directly call the Java's `java.io.ObjectOutputStream` API to write objects to a file.

Listing 24.16 shows some sample code that checks to see whether a CFC exists in the session scope. If the CFC isn't found in the session, the code checks to see whether a serialized copy of the component exists in a file and loads that. Finally, if the CFC is neither in the session nor on the disk, the CFC is instantiated and written out as a new serialized copy.

Listing 24.16 `serialize.cfm`—Serializing a CFC

```

<!---
  FileName: serialize.cfm
  Author: Ken Fricklas (KF)
  Purpose: Implement a distributed, serialized system
  -->
<cfapplication sessionmanagement="yes" name="serialdemo">
<cfif not isdefined("session.cfcDirector")>
  <!--- check to see if we've got a copy --->

```

Listing 24.16 (CONTINUED)

```
<cftry>
  <cfset fileIn = CreateObject("java", "java.io.FileInputStream")>
  <cfset fileIn.init(expandpath("./serialized_director.txt"))>
  <cfset objIn = CreateObject("java", "java.io.ObjectInputStream")>
  <cfset objIn.init(fileIn)>
  <cfset session.cfcDirector = objIn.readObject()>
  Read!
  <cfcatch>
    <!-- no copy to load, create it -->
    <cfset session.cfcDirector = createObject("component", "director")>
    <!-- save it -->
    <cfset fileOut = CreateObject("java", "java.io.FileOutputStream")>
    <cfset fileOut.init(expandpath("./serialized_director.txt"))>
    <cfset objOut = CreateObject("java", "java.io.ObjectOutputStream")>
    <cfset objOut.init(fileOut)>
    <cfset objOut.writeObject(session.cfcDirector)>
    Written!
  </cfcatch>
</cftry>
</cfif>
<cfoutput>
  #session.cfcDirector.showPerson()#
</cfoutput>
```

NOTE

ColdFusion 9 adds a new attribute, `serializable`, to the `<cfcomponent>` tag. If this is set `false`, only the component will be serialized, and any local variables (in the `variables` or `THIS` scope of the component) will not be written, giving you a “clean” copy. This can be used to pass program logic from one machine to another, such as serializing a component, passing it via an HTTP request, and reassembling it on the far side and running the logic there on another copy of ColdFusion.

This page intentionally left blank

INDEX

- # (hatch symbol) comment indicator, 485
 - % (percent) wildcard, 360–361
 - * (asterisk)
 - metacharacter (RegExp), 479–480
 - for wildcard searches, 359
 - ? (question mark)
 - quantifier (RegExp), 478
 - for wildcard searches, 359
 - @ (at symbol)
 - stored procedures and, 404
 - in XPath, 534
 - [] (square brackets)
 - to restrict XPath expressions, 534–535
 - specifying character classes with, 475
 - \ (backslash symbol), 473
 - ^ (carat symbol), 475
 - { } (braces) in scripts, 495, 509
 - | (pipe character) in RegExp, 474, 481
 - + metacharacter (RegExp), 478–479
 - = (equals sign), 65
- A**
- access controls, CFCs and, 88–89
 - Access, Microsoft, E31
 - accordions (Ajax), 249–250
 - ActionScript value object (AIR), 329
 - Add Watch Expression feature, E79, E92
 - addresses (globalization), E24–E25
 - Administrator, ColdFusion. *See* ColdFusion Administrator
 - Adobe Flex. *See also* Flash Builder
 - backend, 120
 - ColdFusion and, 281–284
 - ColdFusion Flex services, 293–294
 - ColdFusion-powered Flex applications, 284–286
 - configuration settings, 296–299
 - Data Management service, 295–296
 - data messaging in, 294–295
 - data translation, 286–289
 - debugging applications, 299–302
 - Flex Server option (FlashBuilder4), 304
 - invoking CFC methods, 292
 - result and fault methods, 291
 - RPC requests, 290–292
 - Adobe LiveCycle Designer, 208–209
 - advanced custom tags, 42–43
 - AIR applications, 323–334
 - AIR library, 282
 - AIR ORM feature, 325
 - CF service CFC, building, 325–328
 - clients for CF applications, 323
 - conflicts, managing, 333–334
 - data, loading, 331–332
 - data, saving and editing, 332–333
 - data, synchronizing with server, 333
 - Data Persistence Library and, 323
 - Flash Builder project, configuring, 324
 - vs. LCDS, 323–324
 - syncManager, initializing, 329–331
 - value objects, building, 328–329
 - Ajax (Asynchronous JavaScript and XML), 243–280
 - Ajax import examples (listings), 277
 - AjaxLink, loading content with, 268–269
 - AjaxProxy (listings), 271–273
 - <CFAjaxProxy>, 271–274
 - debugging, 278–279
 - functions (JavaScript), defining in loaded pages, 278–279
 - functions, running on page load, 278
 - JavaScript Ajax libraries, importing, 276–278
 - JSON and, 274–276
 - layout controls. *See* layout controls (Ajax)
 - populating controls dynamically, 263–271
 - security options, 279–280
 - also attribute, algorithm for, 188
 - alias attribute, 288–289
 - alternation metacharacter type, 474, 481
 - AMF binary protocol, 282, 294
 - AND searches, 359
 - Apache Derby database, 420
 - API, JavaScript, 269–271
 - APPLICATION scope, storing CFCs in, 75–76
 - applications. *See also* AIR applications
 - Application.cfc, specifying locations with, 28
 - DDX processing application, creating, 197–198
 - email application example, 390–392
 - internationalizing (I18N), E3
 - per-applications settings, enabling, 99–100
 - POP3 example, 389–390
 - arguments
 - CFC argument type checking, 101
 - of functions, 4
 - functions and (scripting), 501–502
 - javascriptPopupLink(), 19
 - JVM, setting, 117
 - reFind() function syntax, 459
 - replaceAll() function (RegExp), 466
 - script statements and, 494–496
 - sync method, 326
 - array and structure functions (XML), 521
 - Array Loops (scripting), 498–499
 - ArrayCollection object type, 286
 - assembler CFC, 313–314
 - asynchronous requests/callbacks (Flex), 285
 - AsyncToken dynamic class, 292
 - attacks on servers, settings for, 103–104
 - attributes
 - attributeCollection, 40–41, 502
 - ATTRIBUTES scope, 29–31, 33, 139
 - <cfargument> tag, 7, 55
 - <cfcatch> tag, E48
 - <cfcomponent> tag, 54
 - <cffunction> tag, 6, 55

- attributes (*continued*)
 - <cfimage> CAPTCHA, 232
 - <cfimport> tag, 41
 - <cfinvoke> tag, 58–59
 - <cfinvokeargument> tag, 59
 - cfpdf optimization attributes, 187
 - <cfpresentation> tag, 335, 341–342
 - <cfpresentationsslide> tag, 336, 341–342
 - <cfpresenter> tag, 343
 - <cfproccomponent> tag, 405
 - <cfproccresult> tag, 401
 - <cfquery> tag, 152–153
 - <cfreturn> tag, 8
 - <cfschedule> tag, 386–387, 388
 - <cfstoredproc> tag, 399
 - <cfthread> tag, 135
 - declaring with <cfparam>, 30–31
 - defining for custom tag (listing), 29–30
 - establishing default values of, 31–33
 - <HTML> tag (G11N), E43
 - ImageDrawText, 228
 - imageNew() function, 238–239
 - ImageRotate() function, 236–237
 - making some optional (listing), 31–32
 - nonstandard attributes, allowing, 103
 - optional or required, 31
 - position, rotation and opacity (watermarks), 183
 - supplying to custom tag (listing), 30
 - using functions to test for, 32–33
 - variable names as tag attributes, 37
 - XML, 513, 519–521
 - autoCommit function (Flex), 317
 - automated email applications. *See* email
 - autoplay (presentations), 340
- B**
- backreferences
 - altering text with, 469–471
 - defined, 469, 470
 - using in RegEx, 486
 - backslashes (\) (metacharacters), 473
 - Bauer, Christian, 416
 - BIDI (bidirectional) locales, E33–E34
 - bindings
 - AjaxProxy as, 274
 - Bind to Data wizard, 290
 - bindings to load content (Ajax)
 - AjaxLink, 268–269
 - basics of, 266–268
 - JavaScript API and, 269–271
 - BlazeDS, 281–283
 - bookmarks, DDX, 198–199
 - borders, adding to images, 227–228
 - bound parameters, 450–452
 - break statement (scripting), 495
 - breakpoints
 - debugging failures and, E86–E87
 - disabling, E95
 - importing/exporting, E95
 - setting, E82
 - browsers
 - browser tab, debugging and, E94
 - internal, E94
 - requests, E71
 - bubbling up, E65
 - Buddhist calendar, E26
 - building user-defined functions
 - basic steps, 4–6
 - local variables, 8–10
 - saving, 10–13
 - tag syntax, 6–8
 - using, 6
 - business rules/processes, 398
- C**
- caching
 - Cache Template in Request
 - option, 108
 - cached queries. *See* query caching
 - caches, inspecting, 167–168
 - Caching page server settings, 107–109
 - Charting page and, 115
 - data, 164–165
 - functions, 166–167
 - page. *See* page caching
 - calendars (globalization), E25–E28
 - CALL Command, 411–413
 - CALLER scope, 34
 - calling template, returning variables
 - to, 34–37
 - Camden, Ray, 211
 - capitalization, ignoring with
 - reFindNoCase(), 461
 - CAPTCHA tests, 232–235
 - Cartesian coordinate system, 184
 - case sensitivity
 - debugging and, E77
 - reFindNoCase() function and, 461
 - catching exceptions, defined, E48
 - categories, using in searches (listings), 372–373
 - CD collection example (XPath & XSLT), 531–533, 537–555
 - CDATA sections, 522–523
 - <cf_initcap> tag, 25–26
 - <cfabort> tag (error handling), E53
 - <cfajaximport> tag, 277
 - <CFAjaxProxy> tag, 271–274
 - <cfargument> tag, 7, 55
 - CFC methods, invoking, 292
 - CFC service, AIR and, 325–328
 - <cfcache> tag, 159
 - <cfcatch> tag (error handling)
 - adding to basic template, E52–E53
 - basics of, E47–E48
 - for trapping errors, 145
 - working around errors with, E55–E56
 - <cfcollection> tag, 374, 375–377
 - <cfcomponent> tag, 54–55
 - CFCs (ColdFusion Components)
 - accessing via URLs, 65–67
 - basics of, 51–53
 - calendar CFCs, E28–E30
 - distributed (serialization), 92–93
 - error executing CFC-based tag equivalents, 509–510
 - extending CFML and, 3, 23
 - file structure of, 54–56
 - as groups of functions, 56–58
 - holding instance data and. *See* CFCs that hold instance data
 - implementing in script, 491–492, 502–505
 - inheritance and, 84–87
 - instance-based components, 53–54
 - interfaces, defining, 87–88
 - introspection, 62–64
 - logic, separating from
 - presentation, 61–62
 - OnMissingMethod method, 91–92
 - security implementation, 88–91
 - simultaneous requests, 106
 - static components, 53
 - type checking for arguments, 101
 - type checking in, 67–68
 - using in ColdFusion pages, 58–61

- CFCs that hold instance data
 - APPLICATION scope and, 75–76
 - basics of, 68
 - example of, 70–75
 - getter/setter properties, 79–80
 - implicit getters/setters, 80–81
 - initializing, 81–82
 - instance data as properties, 76–79
 - locking and, 82–84
 - SESSION scope and, 76
 - shared scopes and, 82–84
 - storing, 75–76
 - THIS scope and, 69–70
- <cfdiv> tag, 267
- <cfdump> tag, 63–64
- <cfexit> tag, 47–48
- <cfupload> control, 258–259
- <cffinally> tag, 508
- </cfform> template, E51
- <cffunction> blocks, 10, 74–75
- <cffunction> tag
 - syntax, 6
 - using to create methods, 55–56
- <cfhttp>, scripting (example), 502
- <cfimage> tag
 - CAPTCHA attributes, 232
 - manipulating images with, 223–235
 - overview, 221–223
- <cfimport> tag, 41–42
- <cfinclude> tag, 12
- <cfindex> tag
 - indexing files with, 353–355
 - for searches, 361–362, 363
- <cfinvoke> tag, 58–60
- <cfinvokeargument> tag, 59
- <cflayout> tag, 243–246, 442
- <cflayoutarea> options (listing), 246–247
- <cflocation> tag, 432
- <cflock> tag, 139
- <cfmap> tag, 260–261
- CFMAPPING folder, 288
- <cfmediaplayer> tag, 261
- <cfmenu> tag, 252–253
- CFML (ColdFusion Markup Language)
 - extending, 3–4
 - gateway, 133–134
 - in globalized applications, E43
 - scripting overview. *See* scripting
- CFML custom tags, 23–49
 - advanced, 42–43
 - attributeCollection, 40–41
 - ATTRIBUTES scope, 29–31
 - basics of, 23–24
 - CALLER scope, 34
 - vs. CFCs, 52
 - <cfimport> tag, 41–42
 - <cfparam> tag, 31–33, 37–38
 - CustomTags folder, 24–27
 - default values, 31–33
 - extending CFML and, 3
 - installing, 24–25
 - nested, 48–49
 - optional and required attributes, 31
 - paired. *See* paired custom tags
 - placing in current directory, 27
 - search path, changing, 26–27
 - specifying locations, 28
 - types of, 34
 - using, 25–26
 - variable names as tag attributes, 37
 - variables, returning to calling template, 34–37
 - variables, setting dynamically, 38–40
 - writing, 28–29
- <cfmodule> tag, 28
- CFMX UTF-8 resource bundle, E20–E21
- <cfoutput> tag, 21
- <cfparam> tag
 - establishing default attribute values with, 31–33
 - using with type=“variableName”, 37–38
- <cfpdf> tag, importance of, 175
- cfpdfparam tag, merging PDF files with (listing), 178
- <cfpod> tag, 251
- <cfpresentation> tag
 - basics of, 335–339
 - PowerPoint and, 346
 - styling presentations and, 341–342
- <cfpresentationslide> tag, 336, 341–342
- <cfpresenter> tag, 343
- <cfproccparam> tag, 405–408
- <cfproccresult> tag (stored procedures), 401–403
- cfproperty tag, 288–289
- <cfproperty> tag
 - defining properties with, 418–420
 - documenting properties with, 78–79
 - value objects, creating, 328
- <cfquery> tag
 - attributes, 152–153
 - calling stored procedures with, 398, 410–414
 - vs. <cfstoredproc> tag, 414
 - scripting (example), 503–504
 - for searches, 361–362
- <cfqueryparam> tag, 450
- <cfrethrow> tag (error handling), E65–E68
- <cfrethrow> tag, using (error handling), E65–E68
- <cfreturn> tag, 8
- <cfschedule> tag
 - basics of, 385
 - creating scheduled tasks with, 386–388, 393
 - defined, 381
 - modifying, deleting and running scheduled tasks with, 388–389
- CFScript equivalents, 492–494
- <cfscript> tag, 488–490
- <cfsearch> tag, 355, 356–357, 365
- <cfset> tag quoted syntax, 39
- <cfsetting> tag, 170–171
- <cfsilent> tag, 170
- CFSPRYDATASET tag, 536
- cfstart.bat script, 300
- <cfstoredproc> tag
 - vs. <cfquery> tag, 414
 - calling stored procedures with, 398, 399–401
 - <cfproccresult> tag and, 402–403
- <cfthread> tag, 106–107, 140
- <cfthrow> tag (error handling), E68–E70
- CFTOKEN value, 100
- <cftooltip> tag, 256, 265
- <cftry> tag (error handling)
 - adding to basic template, E52–E53
 - basics of, E47–E48
 - <cftry>–<cfcatch> constructs, 507–508
 - nesting <cftry> blocks, E61–E64
 - trapping errors with, 145
 - working around errors with, E56–E57
- <cfwindow> tag, 251–252, 264
- CFX tag, 4, 23

- CFXML, creating documents with, 516–517
 - <channel-definition> XML element, 312
 - channels, Flex server, 298
 - character classes (Regex), 474–477
 - character encoding, E14–E16
 - character references, 522
 - Charting page, 115–116
 - child components, 84–86
 - Chinese calendar, E26
 - Class Path setting (CF Admin), 117
 - classes, defined in OOP, 52
 - CLDR (Common Locale Data Repository), E5–E6
 - CLIENT variables, 109–110
 - clients, AIR (F applications), 323
 - client-side page caching, 159–161
 - code
 - initialization code, 74
 - reusing, CFCs and, 52
 - reusing when saving UDFs, 13
 - stepping through for debugging, E88–E91
 - ColdFusion
 - Adobe Flex and, 281–284
 - Collections page, 118–119
 - Components (CFCs). *See* CFCs (ColdFusion Components)
 - configuring for debugging, E73
 - Data Persistence Library. *See* Data Persistence Library
 - Debugging perspective, E81
 - Enterprise and Developer editions, E74–E75
 - Evangelism Kit PDF file, 185
 - extending, 3, 23
 - Flex services, 293–294
 - line debuggers, 302
 - new internationalization features, E36–E37
 - service CFC, building, 325–328
 - validating XML in, 528–530
 - ColdFusion Administrator. *See also* server configuration
 - basics of, 97–98
 - configuring, E74–E76
 - configuring for debugging, E74–E76
 - Flex settings in, 296–299
 - indexing files with, 352–353
 - options in, 151–152
 - scheduled tasks. *See* scheduled tasks (CF Admin)
 - Timeout Requests feature, E75, E88
 - tools for thread information, 148–149
 - ColdFusion Builder
 - configuring for debugging, E76–E80
 - Debug Configurations page, E79–80
 - Debug Mappings page, E78
 - Debug Settings page., E78
 - opening files and, E89
 - project, debugging, E77
 - ColdFusion Markup Language (CFML). *See* CFML (ColdFusion Markup Language)
 - collation (sorting) functionality (locales), E9–E14
 - collections
 - Collections page, 118–119
 - defined, 118
 - collections (Solr)
 - adding custom fields to, 367–368
 - administering with <cfcollection>, 375–377
 - basics, 351
 - creating, 351–352, 377
 - custom fields, adding to (listing), 367–368
 - defined, 351
 - deleting, 376–377
 - index collections with database data (listing), 362
 - indexing, 352–355
 - maintaining, 374–375
 - optimizing, 376
 - repopulating, 374–375
 - scheduling updates to, 392–393
 - searching multiple, 370
 - columns
 - defining (persistent CFCs), 418
 - indexing multiple search query columns, 365–367
 - indexing with custom fields, 367–370
 - returned by Verity searches, 358
 - comment statements (scripting), 500
 - comments, Regex, 485
 - commit method, 333
 - Common Function Library Project, 21
 - Component Cache, 108
 - conflict resolution
 - adding to forms (Data Management application), 318
 - defined, 295
 - with LCDS, 324
 - conflicts (AIR)
 - detecting, 327–328
 - managing, 333–334
 - Connect to ColdFusion wizard, 290
 - console output for debugging (Flex), 299–301
 - CONTENT-LANGUAGE META tag (HTML), E43
 - CONTENT-TYPE META tag (HTML), E43
 - controls. *See also* layout controls (Ajax)
 - bindings to load content. *See* bindings to load content (Ajax)
 - control options, setting (presentations), 340
 - populating dynamically (Ajax), 263–271
 - “convert” action, 227
 - count() function (XPath), 544
 - CRUD code, 420
 - CSS (Cascading Style Sheets)
 - <cf_initcap> tag and, 25–26
 - in G11N applications, E41
 - custom exceptions, E68–E70
 - custom fields, indexing columns with, 367–370
 - custom tags, CFML. *See* CFML custom tags
- ## D
- data
 - caching, 164–165
 - Data and Services section (CF Admin), 118–120
 - Data Management Services, 295–296, 323–324
 - data messaging, 294–295
 - Data Sources page (CFAdmin), 118
 - data-centric wizards (Flash Builder), 290
 - DataServicesMessaging event gateway type, 319
 - datasource parameters, 400
 - data-transfer object (DTO), 287

- displaying using `getMetaData()` (listing), 64
 - loading (AIR), 331–332
 - saving and editing, 332–333
 - synchronizing with server (AIR), 333
 - translation between CF and Flash, 286–289
 - data grids
 - in Flash Remoting application, 308–309
 - synchronizing, 314–317
 - Data Management application
 - adding conflict resolution to form, 318
 - configuring CF data push destination, 312–314
 - overview, 311–312
 - synchronizing data grid, 314–317
 - Data Persistence Library
 - basics of, 323
 - Flash Builder project, configuring for, 324
 - data types
 - parameter data types, 408
 - type checking in CFCs, 67–68
 - date objects, 286–287
 - date/time formatting (globalization), E25
 - DDX (Document Description XML), 196–208
 - basics of, 196–197
 - elements, 196–197
 - headers and footers, adding, 200–202
 - simple processing application, 197–198
 - style profiles, 203–205
 - table of contents, adding, 198–200
 - text, extracting, 205–208
 - watermarks, adding, 202–203
 - Debug Output Buffer pane, E93–E94
 - Debugger Settings page, E74
 - debugger tool
 - basics of, E71–E72, E80–E81
 - breakpoints, setting, E82
 - code, stepping through, E88–E91
 - Debug Output Buffer pane, E93–E94
 - Expressions pane, E92–E93
 - further features, E95–E96
 - stopping, E94–E95
 - Variables pane, E91–E92
 - debugging
 - Ajax, 278–279
 - applications, 299–302
 - browsing pages for, E85–E88
 - configuring CF. *See* ColdFusion, configuring for debugging
 - debugger tool. *See* debugger tool
 - debugging perspective, switching to, E81–E82
 - failure of starting sessions, E84–85
 - sessions, starting, E82–E85
 - step debugging, E72–E73
 - task execution, 384
 - traditional forms of, E72
 - Debugging and Logging (CF Admin)
 - Code Analyzer page, 125–126
 - Debugger Settings page, 123
 - Debugging IP Addresses page, 122
 - Debugging Settings page, 120–122
 - License Scanner page, 126
 - Log Files page, 124
 - Logging Settings page, 123–124
 - Scheduled Tasks page, 124
 - System Probes page, 125
 - declarations, XML, 512
 - declaring local variables, 8–10
 - default namespaces, 525
 - deleting
 - collections (Solr), 376–377
 - entities (CFCs), 428–429
 - folders from custom tag search path, 27
 - headers and footers (PDF files), 192
 - pages (PDF files), 180
 - records from Solr collections, 375
 - scheduled tasks, 385, 386, 388–389
 - destinations, setting, 297
 - Developing CFML Applications*, E96
 - dir attribute (<HTML> tag), E34, E43
 - directories
 - default ScriptSrc directory, changing, 102
 - of PDF files, merging, 179–180
 - placing custom tags in current, 27
 - displaying output of threads, 144
 - Document page (CF Admin), 116
 - document search tool
 - collections, creating, 351–352
 - collections, indexing, 352–355
 - collections basics, 351
 - search interface, 355–359
 - documents
 - creating with CFXML, 516–517
 - creating with `XmlNew()`, 518–519
 - structure in XML, 511–514
 - dot-syntax naming scheme, E44
 - Do-While Loops (scripting), 498
 - downsampling PDF images (listing), 188
 - drawing
 - images, 237–241
 - text on images, 228–231
 - DTD (Document Type Definition)
 - standard, 526–527
 - dynamic presentations, 338–339
- ## E
- Eclipse Java debugger, 123, E96
 - edit forms, adding to data grid, 309
 - EditableHomePageDisplay (listings), 467–469, 470–471
 - editing data (AIR), 332–333
 - Ehcache project, 168
 - elements
 - DDX, 196–197
 - XML, 513, 519–521
 - email
 - application, building, 390–392
 - finding with RegEx (listings), 460, 462, 464–465
 - Mail Server page (CF Admin), 112–114
 - embedding
 - content (presentations), 340–341
 - PDF forms, 216
 - empty custom tag, 46
 - encryption of PDF files, 194–196
 - <endpoint> tag, 313
 - entities (CFCs)
 - adding functionality to, 448
 - creating/modifying/saving, 420–423
 - deleting, 428–429
 - Director entities (examples), 429–431
 - entity references, 522
 - entityLoad function, 423–426
 - entityLoad() function, 448–449

- entities (CFCs) (*continued*)
 - EntityLoad() method, 325
 - entityNew() function, 420
 - reading, 423–427
 - updating, 427–428
 - error handling
 - basics of, E48–E53
 - <cfcatch> tag, E47–E48
 - <cftry> tag, E47–E48
 - custom exceptions, E69–E70
 - decisions to not handle
 - exceptions, E64–E68
 - error messages, E47
 - errors, catching as they occur, E47
 - errors, causes of, E53–E56
 - example (threads), 146–148
 - exception handling in script, 507–508
 - exceptions, E47–E48
 - site-wide, settings for, 103
 - templates that recover from
 - errors, E59–E64
 - templates that work around
 - errors, E56–E59
 - escape sequences (RegEx), 474
 - event gateway, creating (messaging application), 319–321
 - event scheduling
 - basic concepts, 379–380
 - CF vs. other scheduling, 380–381
 - <cfschedule> tag basics, 385–389
 - creating with CF Admin, 382–384
 - email application, building, 390–392
 - POP3 application, creating
 - scheduled task for, 389–390
 - scheduled tasks. *See* scheduled tasks (CF Admin)
 - scheduling updates to Solr
 - collections, 392–393
 - exceptions. *See also* error handling
 - basics of, E47–E48
 - bubbling up, E65
 - decisions to not handle, E64–E68
 - predefined exception types, E48–E49
 - in scripts, 507–508
 - thread, 144–148
 - execution cycle of paired custom tags, 43–45
 - expressions
 - copying, E95
 - Expressions pane (debugging), E92
 - extending ColdFusion, 3, 23
 - extractimage action (thumbnails), 182
- ## F
- fetch method (CFC service), 325, 331
 - field search operator (listing), 369–370
 - files
 - class files, saving, 108
 - configuration, 102
 - formats, converting, 227
 - indexing with CF Admin, 352–353
 - indexing with <cfindex> tag, 353–355
 - multiple file uploads (Ajax), 258–259
 - reading with XmlParse(), 514–515
 - saving task output to, 384
 - UDFs and, 10–13
 - FilmRotationCFC component, 70–74
 - filtering posted content with
 - reReplace() function, 466–469
 - fire-and-forget initialization (resource bundles), E45
 - Flash Builder
 - ColdFusion line debuggers and, 302
 - configuring for Flash Remoting, 303–304
 - data-centric wizards in, 290
 - project, configuring for Data Persistence Library, 324
 - Flash Remoting
 - basics of, 120
 - building value objects and, 328
 - configuring Flash Builder for, 303–304
 - RPC services and, 282–283
 - simultaneous requests, 105
 - support for value objects in, 287
 - Flash Remoting application
 - adding edit form to data grid, 309–311
 - adding improvements to data grid, 308–309
 - Example1.xml (listing), 307–308
 - ExampleService.cfc (listing), 304–307
 - Flex, Adobe. *See* Adobe Flex
 - flipping (inverting) images, 236
 - flushing cached queries, 157–158
 - folders, adding to custom tag search path, 26–27
 - Font Management page (CF Admin), 115
 - footers
 - adding to DDX applications, 200–202
 - PDF files and, 189–192
 - using DDX vs. cfpdf tag, 200, 202
 - For Loops (scripting), 498
 - For-In (Collection/Structure) Loops (scripting), 498
 - forms
 - accessing CFCs via, 66–67
 - search form page, creating, 355
 - search results page, 356–358
 - search with user interface (listing), 368
 - simple form for collecting input parameters (listing), 405–407
 - full-text searching. *See* searching full text
 - functions
 - caching, 166–167
 - calendar, E28–E30
 - CFCs as groups of, 56–58
 - G11N-relevant CF functions, E37–E39
 - input/output and, 4
 - JavaScript, defining in loaded pages, 278–279
 - RegEx, 458
 - running on page load (Ajax), 278
 - scripting, 500–506
 - user-defined, 505–506
 - using to test for attributes, 32–33
 - FusionDebug debugger, E73
- ## G
- G11N applications
 - better practices, E40–E46
 - ColdFusion G11N functions, E38–E39
 - G11N, defined, E2
 - monolingual/multilingual G11N designs, E41–E42
 - relevant CF tag/functions, E37–E39
 - general-purpose UDFs, 18–21

Generate Details View wizard, 290
 Generate Form wizard, 290
 GeneratedContent variable, 45
 generic G11N applications, E46
 geoLocator CFCs, E4–E5
 getFilm() UDFs, 4–6, 14–16
 GetMetaData() function, 63–64
 getters

- implicit (CFCs), 80–81
- keeping CFC properties clean and, 79–80

 globalization

- addresses, E24–E25
- applications, internationalizing (I18N), E3
- calendars, E25–E28
- CFML in applications, E43
- character encoding, E14–E16
- date/time formatting, E25
- G11N application better practices, E40–E46
- G11N-relevant CF tag/functions, E37–E39
- generic I1N applications, E46
- global Internet use, E1–E2
- HTML-based META tag, E42–E43
- locales. *See* locales
- monolingual/multilingual G11N designs, E41–E42
- new CF internationalization features, E36–E37
- overview, E1–E2
- page display/layout, E33–E35
- resource bundles. *See* resource bundles
- Solr engine for text searching, E35–E36
- terminology of, E2
- time zones, E30–E31
- Unicode capable databases, E31–E33

 GoogleMap API key

- Ajax and, 260–261
- specifying, 103

 greedy matching, defined, 479
 Gregorian calendar, E25–E26

H

headers

- adding to DDX applications, 200–202
- PDF files and, 189–192

- using DDX vs. cfpdf tag, 200, 202

 heap size, setting, 116–117
 Hebrew calendar, E27
 HelloWorld.cfm (listings), 28–32
 Hibernate framework

- Hibernate Query Language (HQL), 449–450
- lazy attribute and, 452–453
- many-to-many relationships, 437–443
- many-to-one relationships, 433–437
- one-to-many relationships, 433–437
- one-to-one relationships, 443–447
- ORM feature, 325–326
- overview, 416
- relationships overview, 432–433
- sessions and, 431–432

 HINTs, introspection and (CFCs), 62–64
 home page (OWS), implementing with RegEx, 466–471
 HTML (HyperText Markup Language)

- vs. Flex applications, 284
- HTML-based character encoding META tag, E42–E43
- HTML-Based Tree (listing), 256–257
- presentations, 348

 HTTP (HyperText Transfer Protocol)

- status codes, setting, 100
- submissions (PDF forms), 213–214

I

I18N (Internationalization), defined, E2
 ICU4J (IBM), E6–E9
 If-Elseif-Else statement (scripting), 497
 images

- adding borders to, 227–228
- adding watermarks to, 231
- CAPTCHA tests and, 232–235
- <cfimage> tag. *See* <cfimage> tag
- creating and drawing, 237–241
- drawing text on, 228–231
- extracting (PDF files), 182
- flipping (inverting), 236

- imageDrawLine() function, 240
- imageDrawRect() function, 239
- ImageDrawText function, 228–230
- ImageFlip() function, 236
- imageNew() function, 238–239
- ImageRotate() function, 236–237
- imageSetDrawingColor() function, 239
- ImageSetDrawingTransparency() function, 231
- resizing, 223–227
- rotating, 236–237

 include processing statement (scripting), 495
 increment option (generator argument), 420
 indexing

- collections (Solr), 352–355
- collections interactively, 374
- collections programmatically, 375
- columns with custom fields, 367–370
- index values (PDF forms), 210
- table data. *See* table data, indexing

 inheritance, 53, 84–87
 init method, 81–82
 initial views (PDF files), 207–208
 initialization code, defined, 74
 initializing resource bundles, E44–E45
 input parameters (stored procedures)

- basics of, 404
- form for collecting (listing), 405–407

 installing custom tags, 24–25
 instance data

- CFCs that hold. *See* CFCs that hold instance data
- defined, 52–53

 instance variables, 74
 instance-based components, 53–54
 instances

- of CFCs, creating, 60–61
- OOB and CFCs, 52–53
- of syncManager class, 330–331
- THIS scope representing, 69

 interactive indexing, 374
 interfaces

- defined, 87–88
- search interface, creating, 355–359

 Internationalization (I18N), defined, E2

introspection, HINTs and (CFCs), 62–64
 IResponder interface, 331
 isDefined() function, 32
 isJSON function, 275–276
 Islamic calendar, E27
 ISO-8859-1 character set, E14
 ISyncManager interface, 326

J

J2EE mode, debugging and, E76
 Japanese Imperial calendar, E27, E36–E37
 Java
 Java IP (InetAddress) Locator project, E4
 Java Persistence with Hibernate, 416
 Java ResourceBundle class, E21–E22
 Java Virtual Machine, 116–117
 Java-style resource bundle, E21
 JVM page, 116–117
 objects, disabling access to, 101
 serializable interfaces, 92
 JavaScript
 Ajax libraries, importing, 276–278
 API, 269–271
 vs. <cfscript>, 490
 javaScriptPopupLink() function syntax, 19
 packaging for UDFs, 19–20
 joining threads, 137–138
 JPEG format, converting to, 231
 JRun threads, 106
 JSON (JavaScript Object Notation)
 basics of, 274–276
 JSON packets, 66
 prefixes, 279
 prefixing serialized, 101–102
 vs. XML, 274–275

K

.keepServerObject() function, 325
 keystore (mail Server page), 112
 King, Gavin, 416

L

L10N (Localization), defined, E2
 lang attribute (<HTML> tag), E43
 language negotiation, E4
 languages (programming), 457

lastResult property (RemoteObject tag), 306
 layout controls (Ajax)
 accordions, 249–250
 basics of, 243–244
 maps, 260–261
 menus, 252–255
 message boxes, 257–258
 multiple file uploads, 258–259
 pods, 250–251
 progress bars, 262–263
 tabs, 247–249
 tooltips, 255–256
 trees, 256–257
 videos, playing, 261–262
 windows, 251–252
 lazy attribute, 452–453
 LCDS data management, 283–284, 311–312, 315
 LCDS nonblocking I/O (NIO)
 polling, 284
 LCDS Server, 296
 libraries
 FilmFunctions UDF Library, 14–15
 JavaScript Ajax libraries, importing, 276–278
 UDF. *See* user-defined functions (UDFs)
 line debugging
 Allow Line Debugging option (CF Admin), E84
 Flash Builder/ColdFusion Builder for, 302
 line numbers and debugging, E82
 stepping over/stepping into, E88–E90
 lines, drawing, 240
 List Loops (scripting), 499
 literals (literal characters), 473
 LiveCycle Data Services (LCDS)
 vs. AIR applications, 323–324
 connecting ColdFusion to, 283
 defined, 281
 enabling, 120
LiveCycle DDX Reference, 196
 LiveCycle Designer, 208–209
 loading content. *See* bindings to load content (Ajax)
 loading data (AIR), 331–332
 local scope, 9
 local variables
 in custom tag, 33
 defined, 8

UDFs and, 8–10
 locales
 basics, E3–E4
 CLDR (Common Locale Data Repository), E5–E6
 collation, E9–E14
 defined, E2
 determining user's locale, E4–E5
 ICU4J (IBM), E6–E9, E11–E14
 locale stickiness, E42
 localization (L10N), defined, E2
 Location Processing functions (scripting), 501–502
 lock processing statement (scripting), 496
 locking access to the THIS scope, 83–84
 logging. *See also* debugging and logging (CF Admin)
 mail settings and, 114
 scheduled tasks, 381
 logic
 defined, 61
 separating from presentation (CFCs), 61–62
 long-form notation, XML, 520
 lookahead matching (RegEx), 485
 looping in CF scripting (listing), 489
 LTR (left-to-right) layout, E33
 Lucene project, 350

M

mail. *See* email
 many-to-many relationships (Hibernate), 437–443
 many-to-one relationships (Hibernate), 433–437
 mappedBy attribute, 444
 Mappings setting (debugging), E77
 maps, working with (Ajax), 260–261
 matches
 finding with reFind(), 459–462
 including with +, 478–479
 lookahead matching, 485
 match modifiers, 484–485
 matches shortcuts, 476
 matching with * metacharacter, 479–480
 minimal matching quantifiers, 480–481
 multiple, 465
 non-greedy matching, 480
 with pipe character (RegEx), 481

- Memory Variables page (CF Admin), 110–111
 - menus, Ajax, 252–255
 - merging PDF files, 178–179
 - message boxes (Ajax), 257–258
 - messaging application
 - configuring CF data messaging destination, 319
 - creating event gateway (listings), 319–321
 - overview, 318–319
 - metacharacters
 - basics of, 473
 - including as literals, 473
 - types of, 474
 - metadata
 - adding to PDF files, 176–177
 - encryption of PDF files and, 196
 - variables (<cfthread>), 140–141
 - methods
 - calling with <cfinvoke> tag, 58–60
 - CFC methods, Ajax security and, 280–281
 - CFCs, and access to shared scopes, 83
 - <cfexit> tag, 48
 - FilmRotationCFC, 70–71
 - IResponder interface, 331
 - keeping properties clean and, 79–80
 - in OOP, 52
 - overriding, 85–86
 - using <cffunction> tag to create, 55–56
 - Microsoft
 - Access, E31
 - Office documents, creating PDF files from, 216–219
 - SQL Server, E31–E32
 - Migrate Verity Collections option, 119
 - minimal matching, 480
 - missing template handler, 103
 - modifiers (metacharacter type), 474, 484–485
 - mojibake (ghost characters), E15
 - monolingual/multilingual G11N designs, E41–E42
 - multi-line arguments (scripting), 495–496
 - multiline mode (RegEx), 482–483
 - multi-line programming statements (scripting), 496–500
 - multiple-argument functions (scripting), 501–502
 - Multiple-File Uploader (listings), 258–259
 - Multiserver edition of CF, E76
 - <mx:Consumer> tag, 295, 320
 - <mx:DataService> tag, 316
 - <mx:HTTPService> tag (REST), 290
 - <mx:method> tag, 308
 - <mx:Publisher> tag, 295
 - <mx:RemoteObject> tag, 290–291, 305
 - <mx:WebService> tag, 290
 - MySQL 4.1, E32
- ## N
- named templates (XSLT example), 549–555
 - named vs. ordinal parameter positioning, 408
 - names
 - attribute names, 30
 - of functions, 71
 - name attributes, CFML tag, 37
 - public method names, 71
 - variable names, 37–38
 - namespaces, XML, 523–526
 - naming conventions, XML, 514
 - negative lookaheads, 485
 - nested custom tags, 48–49
 - nesting <cftry> blocks, E61–E64
 - network monitors, 302
 - New Flex Project wizard, 303
 - new operator, 504
 - nodes
 - ignoring in hierarchy (XSLT), 542–543
 - retrieving array of with XmlSearch(), 535–536
 - selecting with / and // (XPath), 533–534
 - non-greedy quantifiers, 480–481
 - NOT searches, 359
 - NVARCHAR/NCHAR/NTEXT data types, E31
- ## O
- ObjectEquals() function, 327
 - object-oriented programming (OOP), 52–53
 - objects
 - Java, disabling access to, 101
 - local storage of, 324
 - XML, 515, 518
 - ODBC/JDBC CALL Command, 411–412
 - offline applications with AIR, 323
 - on-demand initialization (resource bundles), E44–E45
 - one-to-many relationships (Hibernate), 433–437
 - one-to-one relationships (Hibernate), 443–447
 - OnMissingMethod method, 91–92
 - Oracle, E33
 - <cfproparam> tag and, 405
 - using <cfproresult> with, 403
 - ordinal positioning, 408
 - ordinal vs. named parameter positioning, 408
 - ORM (object relational model) framework
 - AIR ORM feature, 325
 - basics of, 415–416
 - enabling, 416–417
 - Hibernate ORM feature, 325–326
 - ormFlush() function, 432
 - ormGetSession/ormGetSessionFactory functions, 453
 - persistent CFCs. *See* persistent CFCs
 - searching with, 448–452
 - Output function (scripting), 501
 - output parameters (stored procedures), 404
 - overwrite attribute (thumbnails), 180
- ## P
- page caching
 - <cfcache> tag, 159
 - client-side, 159–161
 - flushing page caches, 163
 - page fragments, 164
 - server-side, 161–162
 - using URL parameters, 162–163
 - pages
 - browsing for debugging, E85–E88
 - deleting (PDF files), 180
 - functions (JavaScript), defining in loaded pages, 278–279
 - functions, running on page load (Ajax), 278
 - modifying properties and, 77–78

- page display/layout (G11N applications), E33–E35
 - page parameter setting statement (scripting), 495
 - pages attribute (PDF files), 179
 - using CFCs in, 58–61
 - paired custom tags
 - execution cycle of, 43–45
 - flow, controlling, 47–48
 - GeneratedContent and, 45
 - paired or empty, 46
 - thisTag.HasEndTag, 46–47
 - parameters, stored procedures that take
 - basics of, 403–404
 - multiple records sets and, 408–410
 - ordinal vs. named parameter positioning, 408
 - parameter data types, 408
 - providing parameters with `<cfprocparam>` tag, 405–408
 - parameters, threadJoin, 135
 - parent components, 84–86
 - passwords
 - PDF files, 192–194
 - scheduled tasks and, 383
 - patterns, RegEx and, 456–457
 - PDF (Portable Document Format) files
 - `<cfpdf>` tag, importance of, 175
 - creating, 176–177
 - creating from Microsoft Office documents, 216–219
 - DDX, using. *See* DDX (Document Description XML)
 - deleting pages, 180
 - documents vs. portfolios, 189
 - forms. *See* PDF forms
 - headers and footers, 189–192
 - images, extracting, 182
 - initial view, controlling, 207–208
 - merging, 178–179
 - optimizing, 187–189
 - pages attribute and, 179
 - portfolios, creating, 189
 - protecting, 192–196
 - reading, 177–178
 - support, 175
 - text, extracting, 184–186
 - thumbnails, creating, 180–182
 - watermarks, creating, 183–184
 - PDF forms
 - embedding, 216
 - populating, 208–213
 - submitting, 213–215
 - Perl
 - boundary sequences, 481
 - RegEx and, 455, 458
 - shortcuts for character classes and, 475–476
 - permissions (PDF files), 192–194
 - Persian calendar, E28
 - persistent CFCs
 - creating entities, 420–423
 - deleting entities, 428–429
 - editing Director entities (example), 429–431
 - overview, 417–418
 - properties, defining, 418–420
 - reading entities, 423–427
 - updating entities, 427–428
 - play options, setting (presentations), 340
 - Pods (Ajax), 250–251, 263–264
 - POP3 application, 389–390
 - populating
 - Ajax controls. *See* controls, populating dynamically (Ajax)
 - PDF forms, 208–213
 - repopulating Solr collections, 374–375
 - portfolios (PDF files), 189
 - ports
 - debugging sessions failures and, E85
 - port numbers when debugging servers, E74
 - scheduled tasks and, 384
 - positive lookaheads, 485
 - POSIX-style shortcuts, 475–476
 - postal codes (ZIP codes), E24
 - PostgreSQL, E32
 - PowerPoint
 - converting from, 346–348
 - support (listing), 347
 - prefixes, JSON, 279
 - presentations
 - basic presentation (listing), 337
 - `<cfpresentation>` tag, 335–339
 - converting from PowerPoint, 346–348
 - dynamic presentation (listings), 338–340
 - embedding content, 340–341
 - play and control options, setting, 340
 - presenters, adding, 343–345
 - saving, 345–346
 - styling, 341–343
 - primary keys, defining (persistent CFCs), 418
 - procedure parameters, 400
 - processing application, DDX (listing), 197–198
 - programmatic indexing, 353–355, 375
 - progress bars, displaying (Ajax), 262–263
 - prompt message box (Ajax), 258
 - properties
 - CFCs, keeping clean, 79–80
 - CFCs, overriding, 84–87
 - channel (Flex), 298–299
 - instance data as, 76–79
 - `<mx:RemoteObject>` tag, 290–291
 - OOP, 53
 - persistent CFCs, 418–420
 - XMLNode, 520–521
 - protect-password.cfm (listings), 192–193, 195
 - proxy servers, scheduled tasks and, 384
 - proxytest.cfm (listing), 272–273
 - Publish and Subscribe, 283, 318
- ## Q
- quantifiers (RegEx), 477–480
 - queries
 - display of in Variables pane, E92
 - failed, working around, E56–E59
 - Flex and, 286
 - multiple search query columns, indexing, 365–367
 - Query Loops (scripting), 499
 - query caching
 - basics, 152–154
 - controlling number of records fetched at once, 158–159
 - limiting number of cached queries, 158
 - maximum number of cached queries, 109
 - refreshing programmatically, 156–158
 - usage example, 154–156
 - quoted `<cfset>` syntax, 39
- ## R
- race conditions, defined, 82
 - raising exceptions, defined, E47

- RB Manager (resource bundles), E22–E23
- RDS. *See* Remote Developer Services (RDS)
- Real-Time Messaging Protocol (RTMP), 283, 294, 296
- record sets
 - multiple, and stored procedures, 408–411
 - stored procedures that return, 401–403
- records
 - in search results, 363–365
 - speeding up browsing of, 154–156
- reFind() function (RegExp), 458–462
- reFindNoCase() function (RegExp), 459–462
- regular expressions (RegExp)
 - appearance of, 457
 - backreferences redux, 486
 - basics of, 455–458
 - character classes, 474–477
 - in different languages, 457
 - escape sequences, 486
 - literals and metacharacters, 473
 - lookahead matching, 485
 - match modifiers, 484–485
 - metacharacter types, 474
 - pipe character, 481
 - quantifiers, 477–480
 - RegExp UDFs, 472
 - string anchors, 481–483
 - support for, 457
 - testing pages, 472
 - uses of, 456–457
 - word boundaries, 481
- regular expressions, using
 - matches, finding, 459–462
 - matches, multiple, 465
 - subexpressions, 462–464
 - text, altering with backreferences, 469–471
 - text, replacing, 465–469
 - where to use, 458
- relationships, defining (persistent CFCs), 418
- reMatch() call, finding multiple matches with (listing), 465
- reMatch() function (RegExp), 458
- reMatchNoCase() function (RegExp), 458
- Remote Developer Services (RDS)
 - configuring to secure servers, E75
 - debugging sessions failures and, E84
- RemoteObject tag, 305–306, 308
- repopulating Solr collections, 374–375
- report threads, maximum number of, 106
- Request Queue Timeout page, 107
- Request Tuning page, 104–107
- requests, browser, E71
- reReplace() function (RegExp), 458, 465–469
- reReplaceNoCase() function (RegExp), 458, 465–466
- resizing images, 223–227
- resource bundles
 - basics, E16
 - initializing, E44–E45
 - locale examples, E17
 - tools and resources, E22–E23
 - types of (flavors), E20–E22
 - using, E17–E20
- Responder classes, 331
- REST (REpresentational State Transfer), 282
- Restart Debugger button, E95
- restrictions in XPath expressions, 534–535
- result and fault methods, 291
- Résumé Manager application (example), 217–219
- return value of functions, 4
- returnSubExpressions (RegExp)
 - argument, 459–460
 - getting matched text using, 461–462
- returnVariable attribute, 39
- RMI (Remote Method Invocation)
 - for data management, 120
- root element, XML, 512, 520
- rotating images, 236–237
- RPC (Remote Procedure Call)
 - requests, 290–292
 - services, 282
- RSS feed, caching, 166–167
- RTL (right-to-left) writing systems, E33–E35
- run time, defined, 7
- S**
- saving
 - data (AIR), 332–333
 - presentations, 345–346
 - savecontent processing statement (scripting), 496
 - UDFs, 10–13
- scale attribute (thumbnails), 182
- scheduled tasks (CF Admin)
 - administering, 381–382
 - creating with CF Admin, 382–384
 - deleting, 386
 - modifying, 385
 - pausing, 384–385
 - running, 384
 - Scheduled Tasks page, 380, 382
- scheduler.log file, 381–382, 385
- scopes
 - ATTRIBUTES scope, 139
 - Debugger Scopes setting, E79
 - local scope, 9
 - thread, 138–141
- scripting
 - CFML overview, 487–488
 - <cfscript> tag, 488–490
 - exception handling in script, 507–508
 - functions, defining in script, 505–506
 - implementing CFCs in script, 491–492
 - replacing tags with, 492–494
 - script functions, 500–505
 - scripting <cfhttp> (example), 502
 - scripting <cfquery> (example), 503–504
 - scripting <cfthread>, 135
 - ScriptSrc directory, changing default, 102
 - statements in, 494–500
 - troubleshooting tips, 508–510
- scrollbars, tab layout and, 249
- searching
 - Ajax-based, 267–268
 - with ORM, 448–452
 - search form page (listing), 355
 - search path, changing (custom tag), 26–27
 - search results page (listing), 356–357
 - text with Solr, E35–E36
- searching full text
 - collections, administering, 375–377
 - collections, maintaining, 374–375
 - improving results, 370–374
 - search tool, creating. *See* document search tool
 - Solr search technology, 349–350

- searching full text (*continued*)
 - SQL data, indexing. *See* SQL data, indexing; table data, indexing
- security
 - Ajax options, 279–280
 - CFCs, 88–91
 - PDF files, 192–196
- <select> statements, 411–412
- selecting nodes with / and // (XPath), 533–534
- serialization
 - distributed CFCs and, 92–93
 - SerializeJSON/DeserializeJSON functions, 275–276
- server configuration
 - ColdFusion Administrator, 97–98
 - Data and Services section, 118–120
 - Debugging and Logging section. *See* Debugging and Logging (CF Admin)
 - Event Gateways section, 129–130
 - Extensions section, 127–129
 - packaging and deployment, 132
 - security, 130–131
 - server monitoring, 126–127
 - settings. *See* server settings (CF Admin)
- Server Monitor, CF Admin, 148
- server settings (CF Admin)
 - Caching page, 107–109
 - Charting page, 115–116
 - Client Variables page, 109–110
 - Document page, 116
 - Font Management page, 115
 - Java and JVM page, 116–117
 - Mail Server page, 112–114
 - Mappings Page, 111–112
 - Memory Variables page, 110–111
 - Request Tuning page, 104–107
 - Settings page. *See* Settings page (CF Admin Server settings)
 - Settings Summary page, 117
- servers
 - connections, defining, E77
 - debugging remote, E74
 - local, debugging, E83
 - massive size server attacks, 103–104
 - multiserver deployment and debugging, E76
 - RDS, configuring to secure servers, E75
 - server output buffer pane, E93
 - synchronizing data with (AIR), 333
 - terminating debugging from, E95
- server-side page caching, 161–162
- Service Capture, 286
- services compiler argument (Flash Builder), 304
- services-config.xml file (Flex), 296–299, 303–304
- SESSION scope, 76, 285–286
- SESSION variables, 111, E42
- sessions, Hibernate framework and, 431–432
- setters
 - implicit (CFCs), 80–81
 - keeping CFC properties clean and, 79–80
- Settings page (CF Admin Server settings)
 - CFC type checking, 101
 - CFTOKEN and, 100
 - default ScriptSrc directory, 102
 - global script protection, 102
 - googleMap API key, 103
 - HTTP status codes, 100
 - internal Java objects, access to, 101
 - massive size server attacks, 103–104
 - missing template handler, 103
 - nonstandard attributes, allowing, 103
 - per-applications settings, 99–100
 - prefixing serialized JSON, 101–102
 - site-wide error handler, 103
 - throttle threshold, 104
 - Timeout Requests After checkbox, 98–99
 - VFS, 102
 - watching configuration files for changes, 102
 - white space management, 100–101
- setVariable() function, 39–40
- shared scopes, 82–84
- shortcuts for character classes, 475–476
- short-form notation, XML, 520
- SimpleJavaScriptFunctions.cfm library, 19–21
- single-argument functions (scripting), 501
- slides
 - autoplaying, 340
 - embedding content in, 341
- Solr search technology
 - basics of, 349–350
 - different types of files, searching with, 350
 - engine for text searching, E35–E36
 - scheduling updates to collections, 392–393
 - searching SQL data without, 360–361
 - table data, indexing. *See* table data, indexing
- SOLR Server page, 119
- special characters in XML, 521–523
- spooling
 - mail settings and, 113–114
 - spool interval, setting, 113
- SQL (Structured Query Language)
 - error codes, displaying (listing), E55–E56
 - SQL statements and stored procedures, 397
- SQL data, indexing, 359–370
 - multiple collections, searching, 370
 - searching without Solr, 360–361
 - Solr and table key values, 370
 - table data. *See* table data, indexing
- square brackets ([]) for character classes, 475
- SSL (Secure Sockets Layer), 113
- Start Debugger button, E95
- statements, scripting, 494–500
- static components (Ajax controls), 53
- static content (Ajax controls), 263
- status codes, returning (stored procedures), 404–408
- status metadata variables (threads), 143–144
- step debugging, E72–E73, E88–E91
- stickiness, locale, E5, E42
- stopping Debugger tool, E94–E95
- storage
 - of CFCs in APPLICATION scope, 75–76
 - client variables and, 109–110
 - default, 110
 - of objects, 324
 - UDFs stored in separate files (listing), 75–76
- stored procedures

- calling from CF templates, 398
 - <cfprocparam> tag, providing parameters with, 405–408
 - <cfquery> tag, calling procedures with, 398, 410–414
 - <cfstoredproc> tag, 398–401
 - defined, 397
 - executing, 398
 - invoking, 397
 - reasons to use, 397–398
 - status codes, 404–408
 - that return record sets, 401–403
 - strings
 - anchors (RegEx), 474, 481–483
 - splitting long, 504
 - struct notation, 40
 - structure and array functions (XML), 521
 - style profiles, DDX, 203–205
 - styling presentations, 341–343
 - subexpressions (RegEx), 462–464
 - submissions (PDF forms), 213–215
 - SUPER scope, 86–87
 - suspending threads, 136–137
 - Switch-Case statement (scripting), 497–498
 - sync method (CFC service), 325–328
 - synchronization of data, 295, 333
 - syncManager class
 - initializing, 329–331
 - sync method and, 326
 - syntax
 - <cfcomponent> tag, 54
 - <cffunction> tag, 55
 - <cfinvoke> tag, 58
 - <cfinvokeargument>, 59
 - <cfproperty> tag, 78–79
 - javaScriptPopupLink() function, 19
 - stored procedures and database syntax, 413–414
 - UDF tag, 6–8
 - XPath, 533–535
- T**
- table data, indexing
 - basics of, 361–363
 - columns, indexing with custom fields, 367–370
 - multiple query columns, 365–367
 - record contexts, displaying, 364–365
 - record summaries, displaying, 363–364
 - table of contents
 - adding to DDX applications, 198–200
 - customizing, 200
 - tags. *See also* custom tags, CFML
 - creating UDFs and, 5
 - G11N-relevant CF tag, E37–E39
 - replacing with scripting, 492–494
 - task scheduler. *See* event scheduling
 - templates
 - basic, without exceptions, E50–E51
 - Cache Template in Request option, 108
 - calling stored procedures from. *See* stored procedures
 - maximum number of cached, 108
 - named (XSLT example), 549–555
 - returning variables to calling template, 34–37
 - simple custom tag templates (listing), 28
 - simultaneous template requests, 104–105
 - that recover from errors, E59–E64
 - that work around errors, E56–E59
 - terminateThread() function, 137
 - terminating debugging, E94–E95
 - testing
 - <CF_HelloWorld> custom tag (listing), 29
 - DDX test application (listing), 206
 - for existence of tag attributes, 32–33
 - RegEx testing pages, 472
 - text
 - altering with backreferences (RegEx), 469–471
 - drawing on images, 228–231
 - extracting (DDX), 205–208
 - extracting (PDF files), 184–186
 - getting matched text, 461–462
 - removing based on RegEx (listing), 467
 - replacing (RegEx), 465–469
 - searching with Solr, E35–E36
 - THIS scope
 - basics of, 69–70
 - locking access to, 83–84
 - variable, 28
 - thisTag.GeneratedContent, 45
 - thisTag.HasEndTag, 46–47
 - threads
 - administrator tools for, 148–149
 - CFML gateway compared to, 133–134
 - <cfthread> example, 142–143
 - <cfthread> tag, 134–135
 - displaying output of, 144
 - ending, 137
 - error handling example, 146–148
 - joining, 137–138
 - mail delivery threads, 113–114
 - maximum number of, 106–107, 115
 - starting, 136
 - status metadata variables, 143–144
 - suspending, 136–137
 - techniques for generating/controlling, 149–150
 - thread exceptions, handling, 144–148
 - thread metadata, 140–141
 - thread scopes, 138–141
 - throttle memory/setting, 104
 - thumbnails, creating (PDF files), 180–182. *See also* images
 - time zones (globalization), E30–E31
 - timeouts
 - increase maximum request (debugging), E75
 - page timeouts, debugging and, E87–E88
 - responses from email servers and, 113
 - scheduled tasks and, 383
 - setting values for variables, 111
 - Timeout Request setting (CF Admin), 149
 - Timeout Requests After checkbox, 98–99
 - timing for requests, setting, 107
 - timesheet (sample PDF form) (listing), 209
 - timing of event scheduling, 383–384
 - tooltips (Ajax), 255–256, 265–266
 - toXML component, 211
 - transport level security (TLS), 113
 - trees
 - Ajax, 256–257
 - defined, 256
 - TruncateQuote custom tag, 44–45
 - try-catch handling, 499–500, 507
 - type checking in CFCs, 67–68, 101

type="variableName", using with
<cfparam> tag, 37–38

U

UDFs. *See* user-defined functions (UDFs)
 Unicode, E14–E16, E31–E33, E45
 URLs (Uniform Resource Locators)
 accessing CFCs via, 65–67
 parameters, caching pages with,
 162–163
 resolving for tasks, 384
 scheduled tasks and, 383
 user-defined functions (UDFs)
 vs. CFCs, 52
 CFML, extending, 3–4, 23
 creating/using (listing), 12
 defined, 505
 general-purpose UDFs, 18–21
 input/output and functions, 4
 RegEx, 472
 UDF libraries, 13–18, 21
 UDF libraries, sharing, 21
 usernames and passwords, 383
 user's locale (globalization), E4–E5
 UTC (Universal Time Coordinate)
 format, 286–287, E40
 UTF-8 encoding, E16, E21

V

validating XML, 526–530
 value objects
 building (AIR), 328–329
 returning, 287–289
 values
 <cfparam> to establish default,
 31–33
 outputting attribute values, 29–30
 var keyword, 138, 506
 variables
 CFCATCH, E53–E54
 <cfthread> metadata variables,
 140–141
 changing on the fly (debugging),
 E93
 copying, E95
 local, in custom tags, 33
 local, UDFs and, 8–10
 making private to functions, 506
 returning to calling template,
 34–37
 setting dynamically, 38–40

variable assignment statement
 (scripting), 500
 variable names as tag attributes,
 37
 Variables pane, debugging and,
 E91–E92
 VARIABLES scope, CFCs and,
 79
 Verity K2 Server page, 119
 Verity search engine, 350, 356–358
 VFS (virtualized file systems), 102
 videos, playing (Ajax), 261–262
 views, initial PDF, 207–208
 virtualized file systems (VFS), 102

W

watermarks
 adding to DDX applications,
 202–203
 adding to images, 231
 creating (PDF files), 183–184
 uses of, 182–183
 WDDX packets, 66
 Web server paths, caching, 109
 Web Services
 aliases, creating, 119
 connecting to CF servers with,
 282
 simultaneous requests, 105
 Web sites, for downloading
 geoLocator CFCs, E5
 RB Manager, E23
 UDF libraries, 506
 Web sites, for further information
 address formats, E25
 address styles (global), E40
 Adobe LiveDocs, 221
 calendars, E25–E30
 ColdFusion applications, xxvi
 ColdFusion Evangelism Kit PDF
 file, 185
 collation (Unicode Consortium),
 E10
 Common Function Library
 Project, 21
 encryption of PDF files, 194
 FusionDebug, E73
 IANA page on character sets, E15
 Java IP (InetAddress) Locator
 project, E4
 Java serializable interfaces, 92
 Lucene project, 350
 multiserver deployment, E76

PDF files, 175
 PDF portfolios, 189
 resource bundle management
 tools, E23
 RTL page layout, E34
 toXML component, 211
 Unicode, E15
 Unicode Collation Algorithm
 (UCA), E14
 Unicode Consortium, E6
 Wolfram Mathworld articles,
 236–237
 XML, 530
 XPath and XSLT resources,
 555–556
 WEB-INF.cftag.component, 85
 WEB-INF/flex/services-config.xml
 file, 305
 While Loops (scripting), 498
 white space management, 100–114,
 168–171
 wildcards
 RegEx, 455–456
 searching with, 359
 windows
 Ajax, 251–252
 Ajax-loaded window (listing), 264
 Windows-1252 character set, E14
 word boundaries (RegEx), 481
 writeToBrowser() element, 236

X

XDП files, defined, 212
 XML (Extensible Markup Language).
See also DDX (Document
 Description XML)
 creating documents with
 CFXML, 516–517
 creating documents with
 XmlNew(), 518–519
 document structure, 511–514
 documents, creating from
 database content, 516
 elements and attributes, 513,
 519–521
 vs. JSON, 274–275
 namespaces, 523–526
 naming conventions, 514
 PDF Form Data (listing), 211
 populating PDF form with
 (listing), 211
 reading files with XmlParse(),
 514–515

- resources for, 530
- schemas, 526–530
- special characters in, 521–523
- transforming XML content with
 - XSL, 542
- validating, 526–530
- XmlFormat() function, 517, 522
- XmlNew() function, 518–519
- XmlParse() function, 514–515, 528
- XmlSearch() function (XPath), 535–536
- xmlText property, 519
- XmlTransform() function (XSLT), 542
- XmlValidate() function, 528–529
- XPath
 - CD collection example, 531–533
 - overview, 531
 - resources for, 555–556
 - syntax, 533–535
 - XmlSearch() to retrieve array of nodes, 535–536
- XSLT (Extensible Stylesheet Language for Transformations)
 - ignoring nodes in hierarchy, 542–543
 - overview, 536–537
 - resources for, 555–556
 - stylesheet, creating, 537–541
 - transformation example using <xsl:if>, 543–544
 - transformations with
 - XmlTransform(), 542
 - using named templates, 549–555
 - <xsl:apply-templates>, 541
 - <xsl:attribute> tag, 547
 - <xsl:choose tag, 544–545
 - <xsl:element> tag, 547–549
 - <xsl:element> to create dynamic HTML elements, 547–549
 - <xsl:for-each> loop construct, 545–547
 - <xsl:if> tag, 543–544
 - <xsl:output> tag, 540
 - <xsl:stylesheet> tag, 540
 - <xsl:template> tag, 540
 - <xsl:text> tag, 545
 - <xsl:transform> tag, 540
 - <xsl:value-of> tag, 540, 541