Adobe

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

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

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

This page intentionally left blank

CONTENTS AT A GLANCE

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

This page intentionally left blank

CONTENTS

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

IN THIS INTRODUCTION

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 multithreaded 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.forta.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

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, object 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
`hint`	Optional. What your component does, in plain English (or whatever language you choose, of course). I recommend that you provide this attribute.
`displayName`	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 `displayName` to make its purpose clear.
`output`	Optional. See `output` under `<cffunction>` below. Only affects any code not inside a `<cffunction>`.

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
`name`	Required. The name of the function (method), as discussed in Chapter 23.
`hint`	Optional. A description of the method.
`displayName`	Optional.
`returnType`	Optional.
`returnFormat`	Optional. The format in which the data should be returned when accessed remotely. By default, all data is returned in WDDX format, unless `returnType` is XML. You can specify WDDX, JSON (for JSON format, used by Ajax), or plain (for no formatting).
`access`	Optional. This attribute defines where your method can be used. See the "Implementing Security" section below for more information.
`roles`	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.
`output`	Optional. If `false`, acts like the entire function is within a `<cfsilent>` tag. If `true`, acts like the entire function is within a `<cfoutput>` tag. If not set, acts normal; variables being output must be in `<cfoutput>` 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
`component`	The name of the component, as a string (the name of the file in which you saved the component, without the `.cfc` extension) or a component instance.
`method`	The name of the method you want to use.
`returnVariable`	A variable name in which to store whatever value the method decides to return.
(method arguments)	In addition to the `component`, `method`, and `returnVariable` attributes, you can also provide values to the method's arguments by providing them as attributes. For instance, the `listFilms()` method from Listing 24.1 has an optional argument called `searchString`. To provide a value to this argument, you could use `searchString="Saints"` or `searchString="#FORM.keywords#"`. You can also provide arguments using the separate `<cfinvokeargument>` tag (see Table 24.5).
`argumentCollection`	Optional. This attribute lets you provide values for the method's arguments together in a single structure. It works the same way as the `attributeCollection` attribute of the `<cfmodule>` 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
`name`	The name of the argument as specified in the arguments of the method
`value`	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

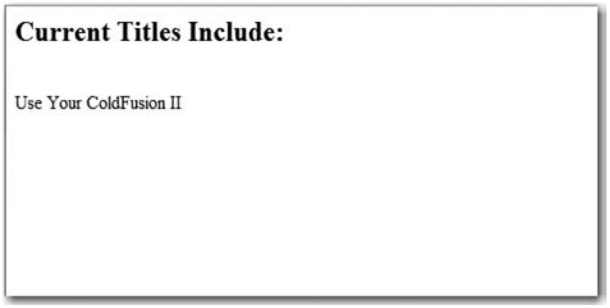
```
<!---
 Filename: UsingFilmSearchCFC.cfm
 Author: Nate Weiss (NMW)
 Purpose: Uses the FilmSearchCFC component to display a list of films
--->

<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.



Current Titles Include:

Use Your ColdFusion II

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.

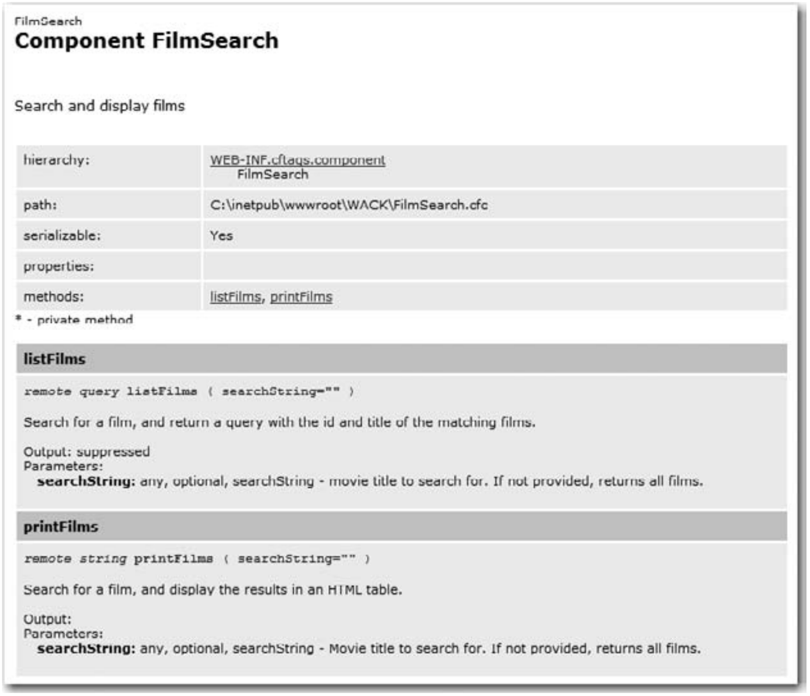
Introspection and HINTs

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.



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.

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

`cfdump` and the `GetMetaData()` Function

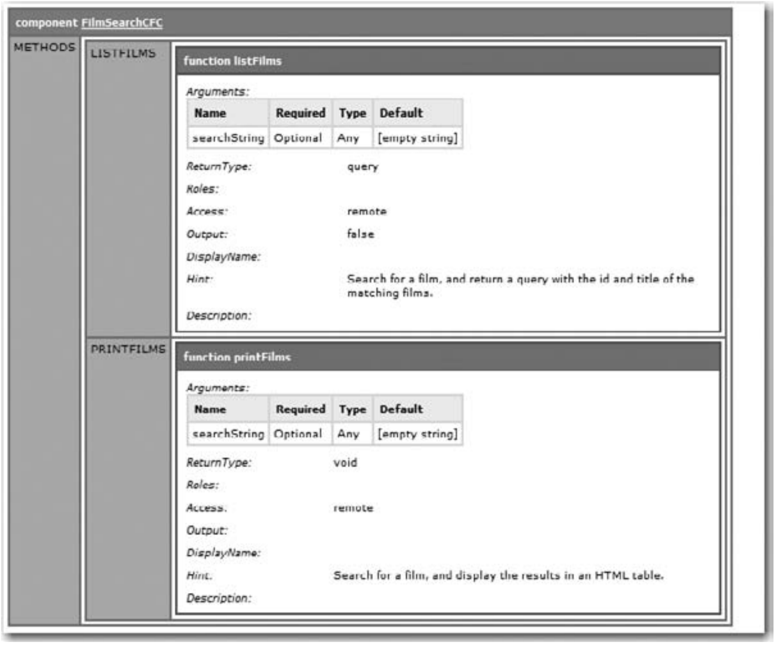
You can dump a component with `cfdump`. For example, you can dump `FilmSearchCFC` as shown here:

```
<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.



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]

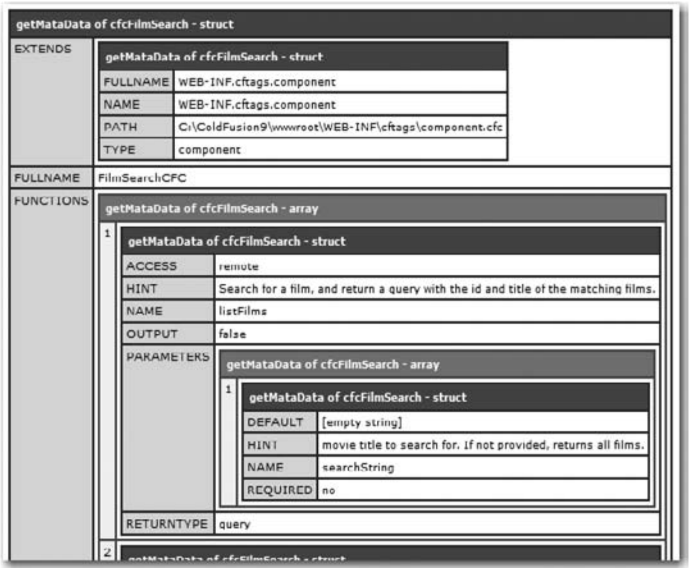
ReturnType:	void
Roles:	
Access:	remote
Output:	
DisplayName:	
Hint:	Search for a film, and display the results in an HTML table.
Description:	

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.



```
getMataData of cfcFilmSearch - struct
EXTENDS     getMataData of cfcFilmSearch - struct
            FULLNAME  WEB-INF.cftags.component
            NAME      WEB-INF.cftags.component
            PATH      C:\ColdFusion9\wwwroot\WEB-INF\cftags\component.cfc
            TYPE      component
FULLNAME    FilmSearchCFC
FUNCTIONS   getMataData of cfcFilmSearch - array
            1  getMataData 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  getMataData of cfcFilmSearch - array
                           1  getMataData 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
```

Figure 24.4 The result of the `getMetaData()` function.

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.cfc?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`.

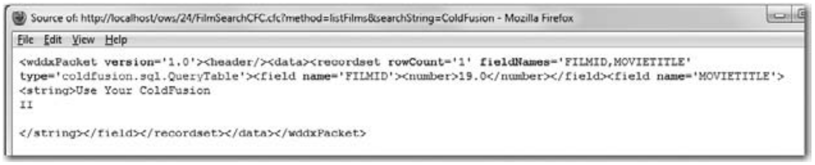
NOTE

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
```


Source of: http://localhost/ows/24/FilmSearchCFC.cfc?method=listFilms&searchString=ColdFusion - Mozilla Firefox
File Edit View Help

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

</string></field></recordset></data></wddxPacket>
```

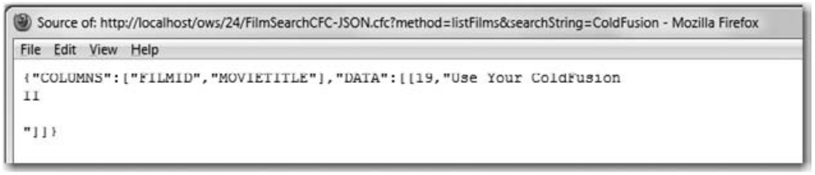
Figure 24.5
A WDDX packet returned by running a component remotely.

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!


Source of: http://localhost/ows/24/FilmSearchCFC-JSON.cfc?method=listFilms&searchString=ColdFusion - Mozilla Firefox
File Edit View Help

```
{"COLUMNS":["FILMID","MOVIETITLE"],"DATA":[[19,"Use Your ColdFusion
II

"]]}
```

Figure 24.6
The JSON response returned by component.

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 `ParseDateTime()`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 `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` where each `x` 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 `xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxxx` where each `x` 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 `<cfobject>` 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
`currentFilmID()`	Returns the ID number of the currently featured film. Because this method uses `access="Private"`, it can only be used internally within the `FilmRotationCFC`.
`isFilmNeedingRotation()`	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.
`rotateFilm()`	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 `isFilmNeedingRotation()` to find out if the current film has expired. If so, it sets the current film to be the next film in the rotation.
`getCurrentFilmID()`	Rotates the current movie (if appropriate) and then returns the currently featured film. Internally, this function calls `rotateFilm()` and then returns the value of `currentFilmID()`. This is a public method.
`getCurrentFilmData()`	Returns the title, summary, and other information about the currently featured film. Internally, this function calls `getCurrentFilmID()` and then returns the information provided by the `GetFilmData()` method of the `FilmDataCFC2` component. This method is included mainly to show how to call one component's methods from another component.
`randomizedFilmList()`	Returns a list of all `FilmID` numbers in the `ows` database, in random order. Internally, this uses the `listRandomize()` method to perform the randomization.

Table 24.7 (continued)

METHOD	DESCRIPTION
`listRandomize(list)`	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 `access="Private"`, it can only be used internally within the `FilmRotationCFC`. 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
`name`	The name of the property. This should match the name of the variable in the `THIS` scope that is used within the component's methods.
`type`	The data type of the property, such as `numeric`, `string`, or `query`.
`required`	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 FilmRotationCFCc.cfc with init Method

```
<!---
 Filename: FilmRotationCFCc.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.FilmRotatorc = 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 `<cfinvoke>` 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">
   </cffunction>
</cfinterface>
```

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

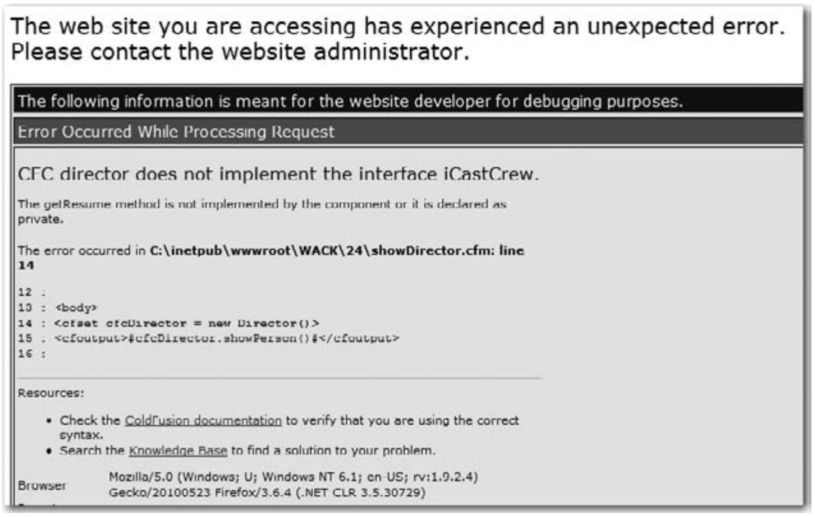
NOTE

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.



```
The web site you are accessing has experienced an unexpected error.
Please contact the website administrator.

The following information is meant for the website developer for debugging purposes.
Error Occurred While Processing Request

CFC director does not implement the interface iCastCrew.
The getResume method is not implemented by the component or it is declared as
private.

The error occurred in C:\inetpub\wwwroot\WACK\24\showDirector.cfm: line
14

12 :
13 : <body>
14 : <cfset cfcDirector = new Director()>
15 : <cfoutput>#cfcDirector.showPerson()#</cfoutput>
16 :

Resources:
  • Check the ColdFusion documentation to verify that you are using the correct
    syntax.
  • Search the Knowledge Base to find a solution to your problem.

Browser   Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.4)
          Gecko/20100523 Firefox/3.6.4 (.NET CLR 3.5.30729)
```

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