PROFESSIONAL COLDFUSION CONTENT FOR THE COLDFUSION PROFESSIONAL

The Fusion Authority Quarterly Update

FALL2006

THE BUILDING BLOCKS OF OBJECT-ORIENTED PROGRAMMING IN CF

FRAMEWORKS DEMYSTIFIED

ALL THIS AND MORE BY YOUR FAVORITE COLDFUSION GURUS!

FALL 2006 $14.95
ISSN 1932-0264
00
02>
9 771932 026000

WWW.FUSIONAUTHORITY.COM

CONTENTS

Fusion Authority Quarterly ■ Vol. 2 ■ Issue 2 ■ Fall 2006

Editorial

Columns

Features

Concepts

Tools

From User-Defined Functions to ColdFusion Components

By Michael Dinowitz

In order to use Object Oriented Programming (OOP) in ColdFusion, you need to understand ColdFusion Components (CFCs). In order to understand CFCs, you need to understand User-Defined Functions (UDFs). These two concepts are at the root of almost every ColdFusion framework, and are the starting points for any use of OOP in ColdFusion.

Some people have never used either UDFs or CFCs, while others use them only because they've been told to use components, but they don't truly understand how these technologies work. I'll be trying to take anyone from knowing nothing about UDFs or CFCs, and build them up to a full understanding of what they can do. In doing so, we'll examine some of the best practices and dogma that have sprung up around these technologies. Only by understanding how things actually work can we know how to use them.

History

User-Defined Functions were first introduced in ColdFusion 5 as a feature of the CFSCRIPT language. This language, designed to be used within the CFSCRIPT tag, is limited in functionality to standard flow control, logic and ColdFusion functions; ColdFusion tags are not available. In addition, because CFSCRIPT is based on JavaScript, those not familiar or comfortable with the syntax were left out. When ColdFusion MX 6 was released, UDFs received a major boost with the inclusion of the CFFUNCTION tag. This allowed people to write UDFs using the full range of ColdFusion tags and functions and provided other advantages as well.

ColdFusion Components were also introduced in ColdFusion MX 6 and were first used by many simply as containers for UDFs. This was a waste of the power of CFCs, but it took time for the community to learn how to use them properly. As understanding of CFCs grew, certain limitations became apparent, certain ideals and dogmas were voiced and many new concepts were introduced to the ColdFusion community (such as object-oriented programming and design patterns). Subsequent releases of ColdFusion have removed many of the early limitations of CFCs and added new features. As of ColdFusion 7, CFCs are seen as a cornerstone of the language.

The major argument now is over the direction in which CFCs should go, with some arguing for more Java-like syntax (interfaces, abstract classes, etc.,) while others are arguing that the move to more advanced OO functionality will only make the language unnecessarily more complex. Since a new version of ColdFusion is in alpha, these debates are more important than ever, and the future of CFCs is being determined as you read.

Article Limitations

This article will focus only on what you need to know in order to use UDFs and CFCs properly. We will not be covering role-based security in CFFUNCTIONS or the use of CFSCRIPT to create UDFs. While CFSCRIPT-based UDFs are still useful and widely used, trying to explain them is a larger undertaking than is possible here. In the CFC section of this article we will be skipping over the topics of web services, remote access to CFCs and component introspection. All of these are important, but they can range far past the scope of this article.

User-Defined Functions (UDFs)

What are User Defined Functions?

User-Defined Functions are composed of two parts: a function definition and a function call. The function definition is composed of a CFFUNCTION tag block, which will contain all of the operations that the function is expected to perform. This tag also has a unique name that will be used when executing the function. A function definition is not executed in line with other code in a ColdFusion template but instead is executed by a function call. In most cases, a UDF function call will look and act exactly like a standard ColdFusion function call and can be used in all the same places/ways. The differences between the two will be discussed later.

Naming a UDF

Every UDF is defined by a name that will be used to execute it. This name must follow a few simple rules:

1. The UDF cannot have the same name as a pre-existing ColdFusion function. There is currently no way of 'overriding' a standard ColdFusion function with a custom one.
2. When a template containing a UDF is processed, the UDF code is compiled into memory and a reference to it is placed in the template's *Variables* scope. For this reason, all of the standard rules for creating a variable name also apply to the UDF name. The UDF name must have as its first character a letter, underscore (_) or dollar sign ($) and may then be followed by any number of additional letters, underscores, dollar signs and/or numbers. As there is no real 'cost' to having a long name, it is a best practice to make the name as descriptive as possible. It is far easier to understand what a function called GETDATE() does than one that is called i().

   Because a UDF is stored in the *Variables* scope, it will overwrite any variable of the same name that was set before the UDF definition. In addition, any variable of the same name set after the UDF will in turn overwrite the UDF, effectively erasing it. This has been known to happen when someone creates a UDF and then uses it to set a variable of the same name.

```
<cfset nextdate=nextdate(now())>
```

   One exception to this is when two UDFs of the same name are defined in the same template. Doing this will always throw a compile-time error, preventing the template from running at all. This will happen no matter how many variables of the same name exist in the template or where they are defined.

3. When a standard variable is created with a period within the variable name, ColdFusion will assume that what comes before the period is a structure that the variable will be assigned to. If the structure does not exist, ColdFusion will automatically create it.

```
<cfset user.name="michael">
   Is the same as
<cfset User=StructNew()>
<cfset User.name="Michael">
```

   This does not apply to UDFs, and a period in a UDF name will cause an error.

Assigning UDFs

There are times when a UDF will have to be assigned to a different variable scope. One such time is when you want to make the UDF available to a Custom Tag. Custom Tags do not have access to the *Variables* scope of a template but they do have access to the *Request* scope. Assigning a UDF

to the *Request* scope not only makes it available to the Custom Tag, but also to any and all code being used by the template, such as CFCs. To assign a UDF to another scope, simply do a CFSET with the UDF name as the value being assigned:

Listing 1: Setting a UDF to the *Request* Scope:

```
1  <cffunction name="nextdate">
2     Function code
3  </cffunction>
4  <cfset Request.nextdate = nextdate>
```

This will create a variable in the *Request* scope containing a reference to the nextdate() UDF. There is no difference between the reference and the original UDF. They do the exact same thing and the only difference between them is in how they are called. NEXTDATE() is called without a prefix while REQUEST.NEXTDATE() uses the *Request* prefix.

Placing the UDF Function Definition

A CFFUNCTION tag can be placed anywhere in the same template as its function call. It does not matter if the tag is placed before or after the call. On the other hand, if a CFFUNCTION is being included in some way, it must always be included before the function call. There are four basic ways in which a CFFUNCTION can be included into a template:

1. **Application.cfm** – A CFFUNCTION in the Application.cfm will be available to any function call in that template or in any template associated with it, such as the template being processed. This is the simplest type of include.
2. **CFINCLUDE** – It is a common practice to place a number of **CFFUNCTION** tags within a separate template and use the **CFINCLUDE** tag to include that template into the requested page. If this is done, the function call must always be after the **CFINCLUDE** tag.
3. **ONREQUESTEND.CFM** – If a UDF is placed within an ONREQUESTEND template, then it will only be available to function calls within the OnRequestEnd template.
4. **Application.cfc** – A note before we begin: A UDF placed within a CFC is referred to as a method of the CFC rather than as a function or UDF. This is just a difference in terminology and should not bother you at all. A UDF=A custom function=a CFC method.

   Because Application.cfc is a component and not a template, there are a few special issues that come up when you want to define a UDF within it. The first issue is that while a UDF can be defined in the component, it will only be accessible by other methods of the component. It will NOT automatically be accessible by the template that was being requested. In order to make the UDF accessible by the requested template you will have to use the ONREQUEST() method of the component to explicitly include the requested template into the component. This makes the *Variables* scope of the component available to the requested template, including any UDFs that are stored within it. If you can't use the ONREQUEST() method, there is another option available. Right after the UDF definition, you can assign a reference to the UDF to the *Request* scope.

Listing 2:

```
1  <cffunction name="nextdate">
2     Function code
3  </cffunction>
4  <cfset Request.nextdate = nextdate>
```

While this requires you to access the UDF as **REQUEST.NEXTDATE()**, it is a small price to pay.

This issue comes up often when people try to replace their Application.cfm templates with the Application.cfc component. The new placement of the UDFs is probably the major stumbling block to this move. For more information on Application.cfc functionality, see *FAQU Issue 1 – Summer 2006*.

Executing a UDF and Passing Parameters

Standard ColdFusion function calls are limited in the way they can pass parameters into a function. All parameters are passed by position, where the physical order of the parameters is important. UDF function calls can pass parameters by position, but they can also pass parameters by name, by data collection and even by using the CFINVOKE tag.

Passing parameters by name is a simple change to the way standard functions pass them. Rather than passing the parameters in a specific order, you assign each parameter a variable name in a *name=value* format. The advantage of this is extra documentation in the UDF function call as well as assigned names for the parameters inside the UDF. For instance:

```
<cfset GetUser('Michael')>
<cfset GetUser(name='Michael')>
```

The first line is a standard pass by position setup, where you simply send the parameter into the UDF and trust that the UDF will handle the naming and other details of the parameter. The second line explicitly says that the parameter being passed in has a variable name associated with it and that this is the name it will have inside the UDF.

Another option for passing named parameters to a UDF is to assign all of them to a structure and then pass the structure to the UDF. The structure is passed by name to the UDF using "ARGUMENTCOLLECTION" as the variable name and the structure name as the value. When a UDF receives a parameter named argumentCollection, it will know that this is actually a collection of variable/value pairs and treat them as if they were named parameters passed to the UDF.

Listing 3: Using argumentCollection to Pass Named Parameters

```
1  <cfset User=StructNew()>
2  <cfset User.Name="Michael">
3  <cfset User.Password="MyPass">
4  <cfset UserID=GetUser(argumentCollection=User)>
```

In addition to the standard manner of calling functions, a UDF can also be called using the CFINVOKE tag. This tag, normally used with components, will not only run a UDF but will allow you to have more control over the parameters being passed. In order to showcase the differences, let's use three different examples. The first will show the outputting of data from a function directly to the screen.

Listing 4: Outputting Data Directly to the Screen

```
1  <cfoutput>#now()#</cfoutput>
2  <cfoutput>#getdate()#</cfoutput>
3  <cfinvoke method=" getdate">
```

The second shows the creation of a variable and its value being set with data returned from a function.

Listing 5: Creating a Variable Using the Results from a Function

```
1  <cfset currentdate=now()>
2  <cfset currentdate=getdate()>
3  <cfinvoke method="getdate" returnvariable="currentdate">
```

The third shows the same thing, but also shows multiple parameters being passed into the function.

Listing 6: Creating a Variable Using the Results from a Function with Multiple Parameters

```
1  <cfset Cleanmail= REReplace(String, '@', '-at-', 'all')>
2  <cfset Cleanmail= ReplaceAt(String, '@', '-at-', 'all')>
3  <cfinvoke method="ReplaceAt" returnvariable="Cleanmail">
4     <cfinvokeargument name="string" value="#String#">
5     <cfinvokeargument name="sign" value="@">
6     <cfinvokeargument name="replacetext" value="-at-">
7     <cfinvokeargument name="scope" value="all">
8  </cfinvoke>
```

As you might expect, the standard ColdFusion function calls and the UDF function calls shown above in Listings 4, 5 and 6 look almost identical. The use of the CFINVOKE tag, on the other hand, looks radically different. The results will be the same in each case, but when data is being passed, the CFINVOKE tag is more explicit in what it is doing. When parameters are passed, they are placed within their own sub-tags, which assign names to them. This means that a function called through a CFINVOKE tag will always pass its parameters by name. While it is possible to place the parameters being passed directly into the CFINVOKE tag itself, this can be very messy when a number of parameters are passed and I strongly suggest not doing it:

```
<cfinvoke method="ReplaceAt" returnvariable="Cleanmail" string ="#String#" sign ="@"
   replacetext ="-at-" scope ="all">
```

The explicit nature of the CFINVOKE tag makes it easier to debug, as there is little question as to where the UDF function call is located, what is being passed and what is being returned. On the other hand, this syntax is not the norm at all, and is rarely seen outside of CFCs. One issue with using CFINVOKE to execute a function is that the tag assumes that the UDF being called is in the *Variables* scope. If a scope prefix is placed with the UDF name in the **method** attribute of this tag, an error will be thrown.

I like using the standard function call syntax when I call a UDF. The only time I use CFINVOKE is when I have to pass three or more parameters. In such a case, I always use the CFINVOKEARGUMENT tags as well. This gives me enhanced clarity as to what's being passed. I find that placing the parameters directly in the CFINVOKE defeats the whole purpose of using the tag and actually makes it much less readable. In addition, I have never found a reason to pass parameters by name or structure in a normal function call.

Assigning/Passing by Reference or Value

It is possible to assign or pass the value of one variable to another. How this assignment works depends on the data contained in the original variable. When it is a simple value or an array the assignment is by value. This means the data is copied to the new variable. When the data is a complex value such as an object reference, structure or query, the assignment is by reference. This means that instead of copying the data from the original variable into the new one, a reference or pointer is assigned to the new variable pointing back to the original. This means any changes to the original will be reflected in the new variable and any changes to the new will be reflected in the original variable.

Listing 7: Passing by Value:

```
1  <cfset name="Michael">
2  <cfset newname=name>
3  <cfset newname="Judith">
```

Listing 7 shows the name variable being assigned to the newname variable. This has the effect of copying the value of the name variable into the newname variable. The newname variable is then assigned a value of "Judith". The change in the newname variable will have no effect on the value of the original name variable.

Listing 8: Passing by Reference:

```
1  <cfset user=StructNew()>
2  <cfset user.name="Michael">
3  <cfset newuser=user>
4  <cfset newuser.name="Judith">
5  <cfoutput>#user.name#</cfoutput>
```

In Listing 8, we have created a structure named user. On line 3, the newuser is assigned a reference to the user structure. We can now think of newuser as an alias of user. If a change is made to user, then the change will be reflected in its alias (newuser). If a change is made to newuser, it will affect the structure contained in user. Therefore, when the name variable in the newuser structure is changed to a value of "Judith", it actually changes the name in the user structure, which will now have a value of "Judith".

The CFFUNCTION Tag

To create a User-Defined Function, we place the code we want executed within a CFFUNCTION tag. This block tag acts as a container for the code and has a number of attributes that relate directly to the calling of the function. Some of these attributes are only useful when a User-Defined Function is used within a ColdFusion Component. Others are not really useful and are usually ignored. The only required attribute is **name**, which is used when executing the function.

Listing 9: Basic UDF creation:

```
1  <cffunction name="GetUser">
2      Function code
3  </cffunction>
```

Other than the **name** attribute, the only other attributes of importance are:

Hint - This attribute is used to document the UDF and it should be used in all UDFs as a best practice. While there is also a **description** attribute, most people ignore it in favor of **hint**.

Returntype – This optional attribute validates the data being returned from the UDF and throws an error if the validation fails. The value of this attribute can either be the name of a datatype that the return data is being tested against or a special validation type. The first special validation type is "VARIABLENAME", which checks if the return data is a string formatted as a valid ColdFusion variable name. The second special validation is used when a CFC reference is being returned from a UDF. The value is the name of a component, which should be the same name as the component in the CFC reference being passed.

The **returntype** only validates the data actually returned by the UDF. If no data is returned, no validation is done and no error is thrown.

This attribute has two additional possible values, neither of which will trigger any validation. The first is "any", which indicates that the UDF may return something, with no restrictions on what that might be. The second is "void", which not only indicates that the UDF will return no data, but actually prevents the UDF from returning data even if it wants to.

Some feel that **returntype** validation is an unnecessary overhead, and therefore avoid using it. Even when no validation is desired, this attribute should be used. We can't go into the exact mechanism, but using **returntype** "any" when data is to be returned is more efficient than not having a **returntype** at all. The same efficiency occurs with the **returntype** of "void" for a UDF that will not return any data.

Output - This is a very important attribute and not using it may lead to problems when a UDF is executed.

When no **output** attribute is defined, any content within the CFFUNCTION block will be outputted as if it were at the location of the function call.

Listing 10: CFFUNCTION with Undefined Output Attribute :

```
1  <cfset test="test value">
2  <cffunction name="showme">
3     this text will be outputted to the screen. #test# will not be evaluated
4     <cfoutput>#test# will be evaluated</cfoutput>
5  </cffunction>
```

When the example above is executed, all of the text inside the CFFUNCTION tag will be outputted exactly as it appears inside the tag. The first #test# will be treated as plain text while the second one will be evaluated just like any other variable in a CFOUTPUT block. In addition, the CFOUTPUT tags will be replaced with whitespace, as is standard for page output.

One thing to pay particular attention to is the whitespace inside the UDF as well as the whitespace generated by ColdFusion tags. This whitespace will not only be outputted when the function is executed, but will also be added to any returned data in the form of a leading space. This space can alter the data being returned from the UDF and, as its origin is not clear, debugging it can be frustrating.

While it sounds like outputting data from a UDF is a bad thing, there are times when you want to do so. Having a UDF output debugging information about itself is a perfect example.

The output effect can be enhanced by adding the **output** attribute to the tag and setting its value to Boolean true (true/yes/!0). This will have the same effect as described above with one alteration: All of the content will be treated as if it was inside a CFOUTPUT block.

Listing 11: CFFUNCTION with Defined Output Attribute:

```
1  <cfset test="test value">
2  <cffunction name="showme" output="true">
3     this text will be outputted to the screen. #test# will be evaluated without a
CFOUTPUT
4   </cffunction>
```

If you do not want any output, and especially not the extra space in the return value, all you have to do is set the **output** attribute to a value of Boolean false (false/no/0). This will prevent any output when the UDF is executed. Even if you had explicitly set content within the UDF for output, it will be suppressed.

Listing 12: Effect of Output Attribute:

```
1  <cfset test="value">
2  <cffunction name="showme"...>
3     #test#
4  </cffunction>
```

Output Attribute Returns	
No Output Attribute	" #test#" (the word #test# as plain text, with a leading space)
Output Attribute Set to True	" value" (the value of the test variable, with a leading space)
Output Attribute Set to False	"" (nothing)

If the **output** attribute is used, it must always have a Boolean value. Any other value, including no value, will cause an error.

There is a disagreement in the community over whether a UDF should output data or not. On one side, people say that a UDF should be limited to data processing and that any data that it would output should instead be returned as a variable and outputted by its caller. On the other side are those people who feel that there is no reason why a UDF should not be used to display content. I belong to the second group, but have a firm rule when writing a UDF that outputs data. Such a UDF should not have a CFRETURN tag and should have little to no ColdFusion code within it. This results in pure user interface-generating UDFs, and pure data-processing UDFs. This clean and clear distinction helps me avoid problems.

The CFARGUMENT Tag

The CFARGUMENT tag is a sub-tag of CFFUNCTION that handles assignment and validation of data passed into the UDF. When used, it must be the first ColdFusion tag inside the CFFUNCTION block. You can use as many CFARGUMENT tags as you want, with the single restriction that each CFARGUMENT tag must have a unique variable name in its **name** attribute, which is required.

The primary purpose of the CFARGUMENT tag is to take data and assign it to the *Arguments* scope of the UDF, while at the same time creating a reference to that variable in the UDF's *Local* scope. In most cases, the data used by a CFARGUMENT will come from a passed parameter.

When the parameter is passed by position, the first parameter will be assigned to the first CFARGUMENT tag, the second to the second, etc. Once a parameter has been assigned, its associated CFARGUMENT tag sets the parameter's value to the *Arguments* and *Local* scopes using a variable name equal to the CFARGUMENT's name attribute.

When parameters are passed by name, ColdFusion will attempt to match the name of a passed parameter with the name of a CFARGUMENT tag. If a match is found, the parameter is assigned to that CFARGUMENT tag and is set to the *Arguments* and *Local* scopes as above.

Parameters without an associated CFARGUMENT tag will be dealt with in a moment. A CFARGUMENT tag without an associated parameter will still create a variable in the *Arguments* and *Local* scopes. If the optional **default** attribute is set, then its value will be set as the variable's value. If no default exists, an empty variable will be created in the *Arguments* scope, with a reference in the *Local* scope. Any data set to this variable location later in the UDF will keep this reference.

In addition to assigning variables, the CFARGUMENT tag can perform two different types of validation on the data being passed. These two validation options are not mutually exclusive. Either or both can be used. The first validation makes use of the optional **required** attribute, which can take a Boolean value. When this attribute is set to "true", the CFARGUMENT tag will throw an error if no

parameter is assigned to it. The value or size of the data contained within the passed parameter does not matter as long as it exists. Having a default attribute set in the tag will prevent the required check from ever failing, thus defeating its purpose.

Each of the examples below is designed to cause an error because a required value was not set. In the first example, no parameter is passed to the UDF and the CFARGUMENT tag requires one.

Listing 13: Error Due to Missing Single Required Parameter:

```
1  <cfset testfunction()>
2  <cffunction name="testfunction">
3     <cfargument name="variable1" required="yes">
4  </cffunction>
```

In this second example, both CFARGUMENT tags have the **required** attribute defined, but only one parameter is passed in. The parameter will be assigned to the first CFARGUMENT tag, leaving the second to throw an error.

Listing 14: One of Two Required Parameters is Missing:

```
1  <cfset testfunction('first value')>
2  <cffunction name="testfunction">
3     <cfargument name="variable1" required="yes">
4     <cfargument name="variable2" required="yes">
5  </cffunction>
```

The third example is much the same as the second, but the parameter is passed in by name. The name of the parameter will be matched to the name of the second CFARGUMENT tag. Because the first CFARGUMENT tag will not be assigned a parameter, an error will be thrown.

Listing 15: One of Two Required Named Parameters is Missing:

```
1  <cfset testfunction(variable2='first value')>
2  <cffunction name="testfunction">
3     <cfargument name="variable1" required="yes">
4     <cfargument name="variable2" required="yes">
5  </cffunction>
```

The second type of validation looks at the data being passed in and checks to see if it fits the criteria set in the **type** attribute. This attribute can have the same values as the **returntype** attribute mentioned above. When data is being passed to a UDF from an outside source such as a form or URL, checking the data type is a very good idea and highly recommended. You should never trust data that is not under your direct control.

The CFARGUMENT tag also has a **hint** attribute that allows you to document what the tag expects passed to it. The few seconds it takes to write a parameter description in this attribute could save you a ton of time later when altering or debugging the UDF.

The *Arguments* Scope

When a UDF is executed, the *Arguments* scope will automatically be created to contain any information passed into the UDF. This scope is private to the UDF and will exist until the UDF has finished processing. The *Arguments* scope can be used as either a structure or an array and any functions that work on either of these two data types will also work on the *Arguments* scope.

The easiest way to add variables to the *Arguments* scope is to use the CFARGUMENT tag. Any parameter passed to the UDF that is not assigned to a CFARGUMENT tag will still be added to the *Arguments* scope, but will not have any of the other advantages that come from the CFARGUMENT tag. If the unassigned parameter is passed in by name, it is added to the *Arguments* scope using that name and is treated as a normal variable in every way. Parameters passed in by position treat the *Arguments* scope as an array, and each parameter is added in positional order.

```
<cfset pretty=makepretty('1/11/71', 'date') >
```

Arguments stored in this manner can be accessed using standard array syntax.

arguments[argumentposition]

Using arguments in this way can be painful, which is why CFARGUMENT tags are so important. If an argument has a name then it is treated like any other variable, with data being read from it or set to it using the *Arguments* scope name as a prefix.

Listing 16: Using the Arguments scope:

```
1  <cfset arguments.newname="Judith">
2  <cfoutput>#arguments.newname#</cfoutput>
```

Finally, once inside a UDF, you can set variables to the *Arguments* scope directly by adding the arguments prefix to the variable name being set.

If you want to access arguments without using the prefix, you will have to make sure they are assigned to the UDF's *Local* scope.

Local Variables

In addition to the *Arguments* scope, there is a special, unnamed *Local* scope available to UDFs. This scope is private to the UDF, allows variables in the UDF to be used without a prefix, and is the first place that ColdFusion will look when trying to evaluate an un-prefixed variable within a UDF. There are two ways of making variables available to the *Local* scope. The first is to use the CFARGUMENT tag and the second is to use a special prefix when setting variables.

When the CFARGUMENT tag is used, it will not only set a variable in the *Arguments* scope, but will also set a reference to that variable in the *Local* scope. These variable settings will be performed even if the CFARGUMENT tag has no data to be set. In such a case, a 'variable space' will be reserved in these scopes with the name of the CFARGUMENT, which can later be assigned a value.

Listing 17: Scope of Variables:

```
1  <cffunction name="testfunction" output="true">
2     <cfargument name="variable1" default="set to Arguments and local scope">
3     <cfargument name="variable2">
4     <cfset arguments.variable3="set to the Arguments scope">
5     <cfset variable2="load variable 2">

6        #variable1# is the same as #arguments.variable1#<BR>
7        #variable2# is the same as #arguments.variable2#<BR>
8        #Arguments.variable3# needs the arguments prefix<BR>

9     <cfset variable1="new value">
10       #arguments.variable1# now has a different value
11 </cffunction>
```

Listing 17 contains two CFARGUMENT tags. The variable names defined in them will be set to the *Arguments* scope with references set to the *Local* scope. If no parameters are passed into this UDF, variable1 will be assigned the default value that is set in the tag. *Variable2*, on the other hand, will have no value and will not be available until a value is assigned to it. When a value is assigned to it, the variable will be accessible from both the *Arguments* scope and the *Local* scope. The third variable set is being assigned directly to the *Arguments* scope and will not be available without using the arguments prefix. Note that I am using the **output**="true" attribute of the CFFUNCTION tag to cause the tag content to be evaluated and outputted to the screen.

If you do not want to use the CFARGUMENT tag to create a local variable, you have the option of using the "var" prefix to set a variable directly to the *Local* scope. This option has a few limits though. You cannot set a variable name that is already being used by a CFARGUMENT tag. A CFSET tag using the var prefix has to be placed at the top of the CFFUNCTION block, immediately after any CFARGUMENT tags. The variable name that is being set cannot contain any periods or references to other variables, such as structures or arrays.

Listing 18: Using var:

```
1  The second CFSET will throw an error.
2  <cfset var holder=structnew()>
3  <cfset var holder.name="Michael">

4  This CFSET will fail.
5  <cfset user.name="Michael">

6  This will succeed, as the variable with a period in its name no longer has a var prefix.
7  <cfset var holder=structnew()>
8  <cfset holder.name="Michael">


9  Local variable setting example:
10 <cffunction name="testfunction">
11     <cfargument name="variable1" default="set to Arguments and local scopes">

12     <cfset var variable2="set to local scope">

13     <cfset variable3="set to Variables scope">
14         ... function code...
15 </cffunction>
```

When the function in Listing 18 is executed, it will create a variable in the *Arguments* scope with a reference in the *Local* scope with the name variable1. If no parameters are passed to this UDF, variable1 will be assigned the value of "set to *Arguments* and *Local* scopes". Then a local variable named variable2 will be set using the var prefix. This variable will be set directly to the *Local* scope with no other reference to it and can never be called with any prefix. The next line will set a variable named variable3 without using the var prefix. This will cause the variable to be assigned to the *Variables* scope of the page the UDF is called from, showing that ANY variable created within a UDF and not set to the *Local* or *Arguments* scopes will be set to the global *Variables* scope.

This may not be what you expect and is the source of many UDF and CFC-based errors. **If you are setting a variable that will only be used within the UDF, you must always make sure that it is set to the *Local* scope.**

The CFRETURN Tag

The CFRETURN tag is used within a CFFUNCTION block to return a variable or value to the UDF function call. A UDF can have multiple CFRETURN tags, each returning a different value based on some programmatic logic or event. When a CFRETURN tag is run, the UDF ceases operation and its local variables are cleared from memory. If a **returntype** is defined in the CFFUNCTION tag, any data passed through a CFRETURN tag will be evaluated against the **returntype** criteria. If this validation fails, an error will be thrown.

A CFRETURN tag can be placed within a CFTRY/CFCATCH block to handle errors in the data being returned. In the example below, the CFRETURN tag is trying to pass a variable that has not been defined. This error will be captured by the CFTRY/CFCATCH block and an alternate CFRETURN will be used instead.

Listing 19: CFTRY and CFCATCH:

```
1  <cfunction name="testfunction" output="true" returntype="struct">
2     <cftry>
3        <cfreturn variable1>
4        <cfcatch>
5           <cfreturn "error">
6        </cfcatch>
7     </cftry>
8  </cffunction>
```

Error Handling

When an error occurs within a UDF, it can usually be handled internally using a standard CFTRY/CFCATCH block. The only cases that cannot be handled internally are validation issues with data being passed to and from the UDF. The three types of validation failures are:

- A CFARGUMENT with the **required** attribute set to "yes" and no data being passed in.
- A CFARGUMENT with the **type** attribute set and data being passed in that is not of the specified type.
- A CFFUNCTION with the **returntype** attribute defined and the data being passed by the CFRETURN tag is not of the specified type.

In all of these cases the error will not exist inside the UDF but will instead be passed back to the UDF function call. In order to handle the error, the function call itself has to be within a CFTRY/CFCATCH block.

UDF variable protection

There are many times where a variable set in a UDF is accidentally set to the global *Variables* scope rather than to the *Local* scope. While people may be scrupulous about variables set using the CFSET tag, they tend not to think about the variables created by ColdFusion tags such as CFQUERY or even CFLOOP. These variables should also be declared as local to the UDF.

Listing 20: Using var with CFQUERY:

```
1  <cffunction name="testfunction">
2     <cfset var userquery="">
3     <cfquery name="userquery" datasource="users">
4        ....query....
5     </cfquery>
6     ... other function code...
7  </cffunction>
```

When a UDF is setting a lot of variables, it can be a pain to make sure they are all set to the *Local* scope. Each new variable needs to be checked to see if it has already been declared as local, and the chances of missing one grows with each new line of code. Hunting down variables accidentally set to the *Variables* scope can be a very time consuming and frustrating operation. There is an easy solution, however, that will allow you to make sure that all of the variables set in a UDF are declared as local, no matter what.

Listing 21: Scope and var:

```
1  <cffunction name="testfunction">
2     <cfset var local=structnew()>
3     <cfquery name="local.userquery" datasource="users">
4        ....query....
5     </cfquery>
6     <cfset local.currentdate=now()>
7        ... other function code...
8  </cffunction>
```

Because the structure named local has been set to the *Local* scope, any variables added to it will also be part of the *Local* scope. The advantage of this is that we do not have to worry about whether a variable has been declared as local or not. As long as it has been assigned to the local structure, it's considered part of the *Local* scope. The disadvantage is that you have to use the prefix "local" to access any variable assigned to the local structure. I prefer to use a local structure when I'm setting multiple variables within a UDF.

Many feel that all variables used within a UDF should be scoped. I agree with this sentiment and the only un-scoped variables in my UDFs are the local variables that were set using the var prefix. Every other variable, even those that have been set with a CFARGUMENT tag and can be used without a scope, will have its scope used. The primary advantage to this is that you always have a visual clue as to what type of variable you're dealing with and where it came from. There is no worry that a variable of the same name in a different scope might accidentally be accessed. This is more important than you may think, as a UDF has access to every variable scope and variable in the template that it is called from. Having an un-scoped variable reading from the wrong location can easily happen and hunting this error down is another time-consuming and frustrating job.

Example

Let us assume that our website has a number of pages that use the same query. If the database changed in some way that affected the query, we would have to find every place where the query is used to update it. One solution would be to put the query into a separate template and include that template everywhere the query is to be used. We will be doing something similar, but instead of including the query into the page, we will place the query into a UDF and return the query data to the UDF function call. In the next part of this article we will expand on the function and make it even more useful.

Listing 22: Using a query variable:

```
1  <cffunction name="QueryLists" returntype="any" output="false" hint="Run a query of the
       lists table and return any data.">
2     <cfargument name="DSN" hint="The datasource needed for the query call">
3     <!--- make the qLists variable local to the UDF. No need for a local structure as only one
          variable is going to be created --->
4     <cfset var qLists="">
```

```
5     <cfquery name="qLists" datasource="#arguments.DSN#">
6        Select ListID, Listname
7        From Lists
8     </cfquery>

9     <cfreturn qLists>
10 </cffunction>
```

When the function in Listing 22 is executed, it will query a database for information and return the query data to the function call. We are not checking the value of the data being passed in, as we know it is a string. We are also not checking whether the data being returned is a query or not. If the CFQUERY is successful, then a variable of the query data type will be returned. The only question is how many rows it will contain (0 or more). If the CFQUERY fails due to a database issue, we will handle it gracefully somewhere else. A database connection failure, which can affect an entire site, is one of the situations that should be handled on a global level rather than in individual queries.

The query in listing 22 returned all of the rows from the table queried. The query in listing 23 will return a specific row.

Listing 23: CFQUERYPARAM:

```
1  <cffunction name="QueryListRow" returntype="any" output="false" hint="Run a query of
       the lists table and return any data.">
2     <cfargument name="DSN" hint="The datasource needed for the query call">
3     <cfargument name="RowID" hint="The row to return from the lists table.">

4     <!--- make the qLists variable local to the UDF. No need for a local structure as only one
          variable is going to be created --->
5     <cfset var qLists="">

6     <cfquery name="qLists" datasource="#Arguments.DSN#">
7        Select ListID, Listname
8        From Lists
9        Where ListID = <cfqueryparam value="#Arguments.RowID#"
            cfsqltype="CF_SQL_INTEGER">
10    </cfquery>

11    <cfreturn qLists>
12 </cffunction>
```

Now that we have a solid grounding in User-Defined *Functions*, we can move onto ColdFusion Components. We will also expand upon our example function and turn it into a full CFC that will handle both the retrieval and caching of a query.

ColdFusion Components

What are ColdFusion Components?

A ColdFusion component is a template with an extension of .cfc. This template usually has a CFCOMPONENT tag, one or more functions (referred to as methods) and usually one or more variables (referred to as properties). None of these are actually necessary and it is possible to have a CFC with all, some or even none of these. We'll deal with this later on.

When you want to use the component, you create an instance of it in memory. This instance can be stored in a variable for future use, stored in a memory scope for long term caching or just used to invoke a method.

Terminology

One of the biggest hurdles when learning to use CFCs is the terminology associated with them. Because CFCs were created to be object-like, many terms from object-oriented programming are used. These terms can be confusing to people who do not have a background in OOP. I'll try to cover the basic terms and their ColdFusion equivalents. You can read more about OOP terminology in Hal Helms' "*Object-Oriented Lexicon*" in this issue.

Child: The CFC using the **extends** attribute of the CFCOMPONENT tag to inherit another CFC (the parent). A child is a more specific version of its parent. Also known as a derived class.

Class: The ColdFusion Component file.

Instance: A unique copy of a CFC held in memory. Multiple instances of a CFC can exist with each being independent of the other.

Instantiation/Instantiating: Creating a variable to hold the CFC so it can be used multiple times in a template.

Invocation/Invoking: Running a method.

Method: A User-Defined Function inside of a ColdFusion Component.

Object: A CFC instance in memory.

Parent: The CFC that is extended when using inheritance. A parent is a more general version of the child. Also known as a base class.

Property: A variable that is part of the CFC. CFCs have two scopes that can contain properties: *Variables* and *This*.

Pseudo-Constructor: Any ColdFusion code in a CFC that is outside of a CFFUNCTION tag. This code will be run when a CFC is instantiated.

Creating and Calling an Example CFC

As stated above, any file with a .cfc extension is automatically a component no matter what its contents. Rather than focus on an extreme example, let's instead focus on a standard one.

Listing 24: A Simple CFC

```
1  <cfcomponent output="false">
2     <cfset Variables.DSN="cfbookclub">
3     <cffunction name="QueryDB" output="false" hint="Query Authors DB and
         return the query">
4        <cfset var Authors="">
5        <cfquery name="Authors" datasource="#Variables.DSN#">
6           Select *
7           from Authors
8        </cfquery>
9        <cfreturn Authors>
10    </cffunction>
11 </cfcomponent>
```

Listing 24 shows the basic elements of a component: A CFCOMPONENT tag block to contain all of the component code, a CFSET tag to create a global property for the CFC, and a CFFUNCTION tag to define a method.

This CFC could be called using the following code:

```
<cfinvoke component="QueryAuthors" method="QueryDB" returnvariable="Authors">
```

The CFINVOKE tag defines the component being used, the method of the component being called and the variable name that the data will be returned to. Now that we've seen a basic component in action, let's delve into the details.

Instantiating and Invoking Components

The term instantiating refers to the loading of a ColdFusion component into memory for use, creating an instance of the component. When the component is instantiated, all of its pseudo-constructors are processed and the component is then ready for use.

If the component was instantiated using the CFOBJECT tag, then a variable will automatically be created to store a reference to the component instance. When the CreateObject() function is used to instantiate a component, a component reference is returned from the function and will have to be manually assigned to a variable. A variable containing a component reference allows the component to be used many times in the same template. In addition, this variable can be assigned to other variable scopes such as *Request* and *Application*. If the variable is assigned to a memory scope such as the *Application* scope, then the component instance will continue to exist in memory until the scope ceases to exist.

If the component was instantiated using the CFINVOKE tag, then the component instance can be used to invoke a single method and is then removed from memory. While this is the normal course of action, there are ways of preventing the component instance from being erased. All of these ways come down to a single operation: storing a reference to the component instance in a variable.

There are a number of other options available when instantiating and invoking components. We won't cover them all here, but you can learn more in this issue's "*Tipical Charlie*" column.

Mapping

In order to instantiate a component, you have to know the component's name and location. If it is in the same directory as the template that will instantiate it, then the component name is all that is needed. This is the filename of the component minus the .cfc extension. A component saved as QueryAuthors.cfc will be instantiated like this:

```
<cfinvoke component="QueryAuthors" method="QueryDB" returnvariable="Authors">
```

If the component is not in the same directory, then the full path to it must be specified along with its name. Each directory in the path should be delimited with a period and there should be no leading period in the path. It is possible to use either forward or back slashes to delimit the path, but this is frowned upon and is not the standard.

```
<cfinvoke component="cfcs.queries.QueryAuthors" method="QueryDB"
  returnvariable="Authors">
```

The example above assumes that the QueryAuthors component is stored in the cfcs/queries directory. ColdFusion will assume that the directory path starts in one of three locations and will try each of them in the following order:

1. Local directory of the calling CFML page
2. Mapped ColdFusion paths
3. Custom tag directories

Pseudo-Constructors

In OO languages like Java, a constructor is called to initialize the variables that will be used by the object. ColdFusion does not have constructors but does have functionality referred to as pseudo-constructors. This is any ColdFusion code in a CFC that is not contained within a cffunction block. When the CFC is instantiated, this code will automatically be executed. In Listing 24 above, the setting of the DSN is pseudo-constructor code.

Constructors and Init()

The one limitation of pseudo-constructors is that they cannot accept variables. To correct this problem, the community has developed a standard to take the place of constructors in CFCs. This standard says that all CFCs should contain a method called init() and that this method should be the first thing called when creating an instance of a CFC. It also says that the init() method should always return the *This* variable scope, which contains a component reference.

Listing 25: Basic init method

```
1  <cffunction name="Init" returntype="any" output="false" hint="Basic init
       method. Returns the This scope to the caller.">
2      <cfreturn This>
3  </cffunction>
```

Listing 25 shows a basic init() method. The **returntype** is set to any to avoid validation. We know what is being returned so any validation here is a waste. Those who want to be strict in their usage of components can set the **returntype** to either the name of the component or the string "web-inf.cftags.component", which is the CFC path to the base component.cfc. The **output** attribute is set to "false" to prevent any output, and a basic **hint** has been added. There is no **access** attribute, causing the method to assume the default of "public". An **access** attribute with a value of "public" can be added for completeness with no overhead.

The *This* scope contains a reference to the component, so when it is returned from the method and stored in a variable, the variable can now refer to the component at any time.

There is a special option available when instantiating a component using the CreateObject() function. This function normally returns a component reference that can then be stored in a variable for future use. It is possible to place a call to one of the component's methods immediately after the CreateObject() function. This will cause the component reference to be used by the method call to invoke the method. The return result of the method, if any, will then be assigned to the variable.

In effect, the method call is hijacking the result of the CreateObject() function. This technique is most commonly used to invoke the init() method immediately after the component has been instantiated. When the method has finished processing, it will return a reference to the component's *This* scope. The *This* scope contains a reference to the component so the end result will still be a component reference being assigned to the variable.

```
<cfset Authors = CreateObject('component', 'cacheobject').Init('cfbookclub')>
```

When the above code is run, the CreateObject() function will return a component reference for the cacheobject component. This component reference will then be used by the init() method call to invoke the init() method contained within the cacheobject component and pass it a parameter of "cfbookclub".

Listing 26: A Sample Component that Uses Init()

```
1  <cffunction name="Init" returntype="any" output="false" hint="Set the DSN and
     return the object reference.">
2    <cfargument name="DSN" hint="(String - Required) The Data Source name to
       be set to the Variables scope">
3    <cfset Variables.DSN=Arguments.DSN>
4    <cfinvoke method="QueryDB">
5    <cfreturn This>
6  </cffunction>
```

The parameter passed to the init method will be assigned to the cfargument tag, which will give it a variable name of DSN and create an *Argument* and *Local* variable for it. The DSN will then be assigned to the component's *Variables* scope, where it will exist until the component is removed from memory. Next, a method called QueryDB that is in the same component will be invoked but will not return any variable. Now that the init method is finished, the *This* scope is sent back to the method call.

The CFCOMPONENT Tag

In Listing 24, the CFC starts and ends with a cfcomponent tag. This tag acts as a container for all other ColdFusion code that is part of the CFC. Any ColdFusion code placed outside of the tag body will cause an error. On the other hand, plain text can be placed before or after the tag and it will be ignored when the component is used.

There are only three attributes of this tag that we really care about. The **output** and **hint** attributes are exactly as described in the UDF section and it is considered best practice to use both. The **extends** attribute is used when inheriting from another component and allows the child component to use the methods and properties of the parent. We will cover this later on when talking about inheritance.

When a cfcomponent tag is used, the CFC automatically inherits from the base component.cfc, which is stored in the "\WEB-INF\cftags" directory. Normally, this component is empty and doesn't even have a cfcomponent tag. It is possible to place code here and have it automatically used by every CFC on the server. This is rarely done, as the practice is not standard and the base component.cfc will be overwritten when a new version of ColdFusion is installed. Those who do place code here tend to limit it to basic methods like init() that should be used by all components.

It is possible to have a CFC that does not use a cfcomponent tag. When you do this, you lose all of the attributes of the tag, including **extends** and **output**. This means that no inheritance is possible and the automatic inheritance of the base component.cfc is eliminated. It also means that there is no output control outside of cffunction tags. Any text that is not contained within a cffunction block will be outputted and all pseudo-constuctor code will generate white-space.

Security

In the UDF portion of this article, we skipped over one attribute of the CFFUNCTION tag that has no use outside of CFCs. This attribute, **access**, is used to control which templates can invoke a method. The default value is "public", which means that any code on the server can invoke the method. The next level of security is called "package", which means that the method can only be invoked by code in the same CFC or from a template in the same directory (.cfc or .cfm). The most secure level, called "private", only allows code within the same CFC to invoke the method.

There is a fourth type of access, called "remote", which is actually more permissive than public. This access allows the method to be invoked through a URL, Form or Web Service, as well as through Flash Remoting. Using remote methods raises a number of issues that have to be dealt with, but they are beyond the scope of this article.

It is considered a good practice to limit access to methods using "private" or "package" when you do not expect to invoke it directly from a ColdFusion template. This allows you to control when and where a method will be used, which for some methods (such as credit card processing) is a MUST. Using "private" and "package" access greatly helps when you have to upgrade or debug the code. The more restricted the access to the code, the easier it is to hunt down a bug.

Some frameworks require that you define an **access** attribute, even if it will only be set to the default value of "public".

Properties - *Variables* and *This*

Properties are the variables defined by the CFC itself. A CFC automatically creates two variable scopes when it is instantiated or invoked. These are the *Variables* scope, which is much like a standard template's *Variables* scope and the *This* scope, which is special to the CFC. Both of these scopes will exist for the life of the component, are accessible to all pseudo-constructors and methods in the component and can store any type of ColdFusion data. Beyond this, the differences between the scopes have to do with "public" vs. "private" access.

Variables

The *Variables* scope is private to the CFC and can only be used from within the CFC. Any variable set inside the CFC that is not declared local to a method or is not set to a specific scope prefix is automatically assigned to the *Variables* scope. In addition, data in the *Variables* scope can be accessed without using a prefix, though ColdFusion will check for a local variable in the method before looking in the *Variables* scope. As with all non-local variables in a CFC, it is a good practice to use the *Variables* prefix, even if it's not needed.

If you want to expose information from the *Variables* scope to the template that invoked the CFC, you will have to create methods to do so. The standard terms for these methods are getters and setters. Getters allow information from the *Variables* scope to be retrieved by the invoking template while Setters allow variables to be set in the *Variables* scope of the CFC.

THIS

The *This* scope differs from the *Variables* scope in that it is public to the template that called the CFC. This means that the calling template can read data from and write data to the *This* scope at will. Because there is no control over who or what can read and write to the scope, there is a dogmatic view that says never to use the *This* scope. Others are a little more lenient and say that the scope should only be used for unimportant data. The problem is that the *This* scope is totally open. There is no control over what code can read data from the scope and what code can write to the scope (even overwriting other data).

If you don't have a reason, don't use the *This* scope. On the other hand, if you have a perfectly good reason to use it, don't let dogma stand in your way.

Encapsulation

The reasoning behind not using the *This* scope is that it breaks encapsulation. This OOP concept says that a component should:

1. Only operate on data passed into the CFC and never assume that data will exist in the environment .

A CFC has access to all of the environmental data that a standard ColdFusion template has. While it could use a *Server*, *CGI* or other variable directly, it is a good idea to only make use of data that has been explicitly sent into the CFC. This helps avoid errors that are based on a false assumption that an environment variable will exist. While some variables, such as standard *Server* scope variables, will always exist, it is better not to assume. The application of this rule is referred to as loose coupling or decoupling, as the CFC is not directly connected to or dependant on another variable, component or anything else. The less a CFC is coupled to another, the easier it is to debug (theoretically) and reuse.

2. Only allow changes to properties that you want changed.

This rule says that the *This* scope should not be used. Because variables in the *This* scope can be added, manipulated and deleted directly without control, it breaks encapsulation. Those who hold by this rule suggest only placing data in the *Variables* scope only, and using Getters and Setters to manipulate the data.

3. Only expose those methods and properties that you want seen.

This rule stresses that if there is a method that should only be called from within the CFC rather than directly, then the method's access should be set to "private" or "package".

For every rule above, there is an exception. The key is knowing if the exception should be used or not. An example of one such exception would be a component designed to manipulate and process CGI information. The component would only be used in a place where CGI information exists.

Caching your Components

Caching a component after it has been instantiated is a simple matter of assigning the variable containing the component reference to a variable in a memory scope. The most commonly used **memory** scopes are *Application* for application-specific components and *Session* for components specific to a particular user. It is also possible to cache a component in the *Server* scope, and this is actually a good idea when dealing with components that will be needed by all applications on the server, such as logging. There is a strong dogma against caching components in the *Server* scope that is a hold-over from the pre-MX days when there were locking issues with memory scopes. This has not been an issue for years, but the dogma persists.

One reason for caching a component is that as long as it exists, all of its properties will exist. This means that if a component has a query stored in its *Variables* or *This* scope, it will continue to exist and not have to be called again. We will use this as the basis for an example that builds on the UDF mentioned in the previous section.

Listing 27: CacheQuery.cfc

```
1  This component will cache the authors in the community example application
2  <cfcomponent output="false">
3      <cffunction name="Init" returntype="any" output="false" hint="Set the DSN and return
           the object reference.">
```

```
4         <cfargument name="DSN" hint="(String - Required) The Data Source name to be set
          to the Variables scope">
5         <cfset Variables.DSN=Arguments.DSN>
6         <cfinvoke method="QueryDB">
7         <cfreturn This>
8       </cffunction>
9       <cffunction name="QueryDB" returntype="void" output="false" access="private"
          hint="Query the Authors table and cache the query">
10        <cfquery name="Variables.CacheQuery" datasource="#Variables.DSN#">
11           Select *
12           from Authors
13        </cfquery>
14      </cffunction>
15      <cffunction name="GetQuery" returntype="any" output="false" hint="Return the query
          to the template">
16        <cfif Not StructKeyExists(Variables, 'CacheQuery')>
17           <cfinvoke method="QueryDB">
18        </cfif>
19        <cfreturn Variables.CacheQuery>
20      </cffunction>
21      <cffunction name="ResetCache" returntype="any" output="false" hint="Resets the
          cached query">
22        <cfinvoke method="QueryDB">
23      </cffunction>
24  </cfcomponent>
```

Listing 28: Test.cfm

```
1  <!--- Example Data Source - cfbookclub (\CFIDE\gettingstarted\community\db\bookclub.mdb) --->
2  <cfif Not StructKeyExists(Server, 'Authors')>
3     <cfset Server.Authors=CreateObject('component', 'CacheQuery').init('cfbookclub')>
4  </cfif>
5  <cfset Authors=Server.Authors.GetQuery()>
6  <cfoutput query="Authors">#authorid# #firstname# #lastname#<BR></cfoutput>
```

The second line of test.cfm (see Listing 28 above) checks if a variable called authors exists in the *Server* scope. If not, it instantiates the CFC, invokes the INIT() function and saves the results of the function into the *Server. Authors* variable. We can then run the GETQUERY() method of the cached CFC and make use of the data. No matter how many times we use the CFC, it will always have the same data in it without having to go back to the database. When we want to refresh the cached query, all we have to do is run the RESETCACHE() method.

While CACHEQUERY is a useful component, there is a problem with it: it is meant for a single use with a single query. In a moment we'll make it a LOT more useful by allowing it to be used by multiple queries without an increase in complexity. How? By using inheritance.

Inheritance

Inheritance is one of the basic concepts of OOP where you have a more general parent component that has a more specific child component. The child has all of its parent's methods and properties as well as its own specific methods and properties. In this way, the child *IS* the parent, but an enhanced version of it, leading to the use of the term "Is A" to describe inheritance relationships.

Inheritance can be one of the hardest OOP concepts to learn because it is usually described in a very abstract manner. What I'm going to do is describe a use of inheritance that is much more practical, which should help you get a better understanding of it.

It should be noted that when a component is inherited, it effectively becomes part of its child. There are no restrictions on the parent using the *Variables* scope of the child and visa versa. Additionally, all methods from either the parent or child are considered part of the same CFC when it comes to methods with a "private" access.

Overriding and Super

When a child inherits from a parent, all of the methods and properties of the parent become methods and properties of the child. If the child already has a method or property of the same name, it is not inherited. This is referred to as overriding, as in the child's methods and properties are overriding those of the parent's.

When a parent's properties are overridden, they can no longer be accessed in any way. They have been totally eliminated from memory.

This is not the case when it comes to the parent's methods. The overridden methods can still be invoked by using the prefix Super before the method call.

```
<cfset oldmethodresult=super.getuser(1)>
```

Even if a parent method has not been overridden, using *Super* will allow you access to it. On the other hand, when *Super* is used, the child methods are rendered inaccessible. In addition, *Super* can only be used from within the child component.

Two different views

Inheritance can be viewed from either the top down or the bottom up.

For the top-down view, we will look at our general CacheQuery component and say that we will have to change the QueryDB() method if we want to use this component to cache anything other than authors. By taking the QueryDB() method and placing it in its own CacheAuthors component, and having that component extend (inherit from) the CacheQuery component, we can now create as many DB caching components as we want, each having the same general functionality but different specific querying functionality.

From the bottom up, we would take our original CacheQuery component and say that we also want to cache poll information. To do so, we will copy everything from the CacheQuery component into a new component called CachePolls. To make things neat, let's also rename the CacheQuery component "CacheAuthors" to keep its name in line with its sibling component. We will then change the CachePolls component so that it is caching poll information.

We now have two very similar components that do very similar functions. If we compare these two operational components we will find that some or most of the methods and properties are the same. We can then move these duplicate methods and properties into a separate, more general CacheQuery component and set that component as the parent of both CacheAuthors and CachePolls.

In the first example we are removing what is unique from the parent component and placing it into more specific children. In the second example we are taking what is similar between two components and moving it into a more general parent. In both cases we have more specific children inheriting from a more general parent.

Listing 29: CacheQuery.cfc

```
1  <cfcomponent output="false" hint="allows for the caching and use of queries that will be
       defined in a child component">
2     <cffunction name="Init" returntype="any" output="false" hint="Set the DSN and return
         the object reference.">
3        <cfargument name="DSN" hint="(String - Required) The Data Source name to be set
            to the Variables scope">
4        <cfset Variables.DSN=Arguments.DSN>
5        <cfinvoke method="QueryDB">
6        <cfreturn This>
7     </cffunction>
8     <cffunction name="GetQuery" returntype="any" output="false"
         hint="Return the query to the template">
9        <cfif Not StructKeyExists(Variables, 'CacheQuery')>
10          <cfinvoke method="QueryDB">
11       </cfif>
12       <cfreturn Variables.CacheQuery>
13    </cffunction>
14    <cffunction name="ResetCache" returntype="any" output="false"
         hint="Resets the cached query">
15       <cfinvoke method="QueryDB">
16    </cffunction>
17 </cfcomponent>
```

Listing 30: CacheAuthors.cfc

```
1  <cfcomponent extends="cachequery" output="false" hint="child component for
     cachequery that handles author data">
2     <cffunction name="QueryDB" access="private" output="false" hint="Query the Authors
         table and cache the query">
3        <cfquery name="Variables.CacheQuery" datasource="#Variables.DSN#">
4           Select *
5           from Authors
6        </cfquery>
7     </cffunction>
8  </cfcomponent>
```

Listing 31: CachePolls.cfc

```
1  <cfcomponent extends="cachequery" output="false" hint="child component for
     cachequery that handles poll data">
2     <cffunction name="QueryDB" access="private" output="false" hint="Query the Authors
         table and cache the query" >
3        <cfquery name="Variables.CacheQuery" datasource="#Variables.DSN#">
4           Select *
5           from Polls
6        </cfquery>
7     </cffunction>
8  </cfcomponent>
```

Type checking parents and children

An interesting aspect of the parent/child relationship comes into play when components are being validated.

Let's use the CachePolls component and invoke the init() method (which is inherited from CacheQuery). Init() returns a reference to the *This* scope, which contains a reference to the component itself. If the init() method had a returntype defined and the returntype was set to CachePolls, then the component would validate as true because the data being returned from init() IS a reference to CachePolls. On the other hand, if we set the returntype to CacheAuthors, we would expect it to fail as they are not the same component.

But what would happen if we set the returntype to CacheQuery? We're calling the CachePolls component, which is not the CacheQuery, but does inherit it. Will the validation work?

The answer is yes. This is because by inheriting the CacheQuery component, CachePolls effectively becomes both itself and CacheQuery. On the other hand, if we were calling the CacheQuery component directly and tried to validate it as a CachePolls component, it would fail. A CachePolls is a CacheQuery but a CacheQuery is not a CachePolls. A Child is its parent plus; a parent is its children minus.

Parting Words

While this article is jam packed with information, it is not the end all and be all of UDF and CFC documentation. Use this as a reference when reading the ColdFusion documentation, the WACK books or some blog. Also, please remember that while I preach against dogma, I also don't want to become a source of it. Just because I say something does not make it right. Go out and test it out yourself. Only by building UDFs and CFCs will you truly understand them.

Additional topics you can look into are:

- Calling CFCs from an URL or Form
- Calling CFCs from Flash
- Using CFCs as Web Services
- Securing CFC methods with role-based security
- Introspection and Metadata
- Building UDFs with CFSCRIPT

Michael Dinowitz is a longtime ColdFusion expert (since early 1995) and is well known for his troubleshooting, experimentation, and ability to take complex topics and break them down into simple elements. He is President of House of Fusion, Publisher of Fusion Authority, and a founding member of Team Allaire/Macromedia/Adobe Community Experts.