

OO TERMS IN A COLDFUSION CONTEXT - My Personal Lexicon

This lexicon available as a [webservice](#)

Jump to Category: [OO](#) [MG](#)

OO TERMS

ABSTRACTION

Consider scenario One:

It's Saturday, and I have a list of chores to do.

1. Throw the laundry into the washer;
2. Do the dishes;
3. Scoop the cat poop out of the litter box;

So, I

1. take the laundry out of the basket in my closet and put it into the washer. I dump in one cup of SudSoClean, set the temp to warm, set the load size to medium, and start the cycle.
2. Move all the dishes out of the sink and onto the counter, wash out the sink, put the drain-stopper in, and fill it with hot soapy water. I put the silverware in first so it can soak the longest, then the plates, and lastly cups. As I wash a dish, I rinse it in the other side of the sink under warm water, then set it on the towel to drain while i finish washing the rest of the dishes.
3. Get a plastic bag from the kitchen drawer and take it to the cat litter box. I get the plastic strainer scooper thingy out of its container, remove the lid to the litter box, and start scooping out little cat tootsie rolls and putting them into the bag. When I'm finished, I add some fresh cat litter, put the lid back on the litter box, and slide it back into place.

Whew! That was hard work, wasn't it boys and girls!

You know what? **I think I'll abstract my chores by adding something between them and myself: my kids!**

Consider scenario 2, with me doing my chores after having abstracted them:

1. I call out to objBrandon and ask him to do the laundry ();
2. I call out to objLilly to do the dishes ();
3. I call out to objSarah to scoop the kitty litter ();

Hey! I'm done with my chores!

Abstraction: *removing the actual work from myself and executing it by commanding someone else to do it.*

It's easy to see why I'd want to abstract my chores. But in an application, WHY would I want to add yet another layer to what may already be somewhat complex?

Because...the way the actual work gets done just might change, and if I have the work abstracted out to individual objects, making those changes is all done in one place.

As an illustration, consider a change to the process for washing my dishes. Now, instead of doing them in the sink, I give in and allow the new process to be rinsing them and stacking them in the dishwasher. I re-write the routine for objLilly, but my call to her to doDishes(), and in fact, anybody else's call to her for the same does not change. Because we've abstracted that chore, the "business logic" associated with it is entirely encapsulated (contained) and found in one place.

In the context of an application, if we DID follow the advice of a peer and "create an abstraction layer for managing persistent variables (such as session variables)", then when the day came that we found a need to stop storing values in session and switch instead to client variables, we don't have to hunt and peck all over our code for every occurrence where a session var gets set; we need only go to our Persistence object and make the needed changes right there.

Abstraction - not such a vague term after all, is it?

ACCESSOR

Anytime you hear someone use this term, just laugh to yourself and say, "Oh, you mean THE GETTER"?

BEAN

Picture if you will...a large, empty table. In the center of the table is a small, single, pinto bean. Okay, you're not too far off from what an OO programming bean is.

You can first off think of a bean as a small, solo object. It's somewhat simple, as objects go, and has no other purpose in life than to store a collection of related values, and the means to get and set them. For example, we have a User bean. It stores the values Firstname, Lastname, Age, Sex, and IQ. We are never allowed to DIRECTLY change or retrieve the bean's values, and can only do so using the bean's SETTER and GETTER methods, like so:

```
<cfscript>
    myUser = CreateObject("component","UserBean");
    userIQ = myUser.getIQ();
    //change the IQ a little...
    myUser.setIQ(userIQ + 10);
</cfscript>
```

What good are beans? You gotta get creative, man! You can create a configuration bean to hold all of your app's configuration settings! That's a cool use. You will definitely be using bean's

when retrieving individual database records. There are a thousand generic uses for them.

Of note is the fact that not everybody calls a bean a bean at all times. Some will choose rather to call it a 'Record' when the bean is being used in conjunction with the DAO and Gateway objects. (but it's really still a bean!)

SO, when you hear the term BEAN, just imagine a very compact, solo object that has nothing but a set of SETTERS and GETTERS for the values it is storing.

BUSINESS LOGIC

So this Controller is sitting at the bar babying a Belvedere Gimlet when he gets a call from his Framework asking him to please provide the value for the current user's next inspection date. A bit perturbed but never one to tarry when the Framework makes a request, he immediately dials his peep, Joe Object, down in the Model district and asks for user 1287's next inspection date. Joe Object quickly does his thing and gives Controller the value he requested, who in turn hands it off to his Framework.

The process that Joe Object went through in order to calculate the user's next inspection date is an example of what is referred to as "Business Logic". It's the process, formula, algorithm, decision tree, methodology, query, magic 8 ball, or any other means used to perform a piece of work that is specific to this application. 'Business Logic' is the answer to the question "How did you get that value?". Consider this dialogue: "Hey, Joe Object, how exactly did you come up with that inspection date?" asks Controller. "Since my job is to encapsulate the processes I use, I can't tell you," Joe Object says. "All I CAN say is that I executed the business logic for that particular request and gave the result to you."

As an aside, anything that you the developer would classify as Business Logic (as defined above), is the PERFECT candidate for inclusion in the CFCs that compose your application's MODEL; in fact, I would go so far as to say that it is mandatory that such code reside within your model. The rule I use (which I gleaned from perusing blogs and mailing lists) when deciding if code is business logic or not is this: "If I had to change frameworks tomorrow and the ONLY thing I can take with me is my Model, is this a piece of functionality I would consider to be uniquely associated with this app?"

CLASS

Which phrasing is correct:

1. "Hey boss, take a look at this object I just wrote!"
2. "Hey boss, take a look at this class I just wrote!"

The answer is number 2. After reading the following definition, hopefully the phrasing in choice 1 will sound strange to you.

The term "CLASS" tends to be misused, or even UNDERused in OO conversations with regard to Coldfusion, and is very often omitted in favor of the term Object. There is, however, an important distinction between the two. Let's illustrate this by asking a simple question that I KNOW you know the answer to.

Can you open a CFC in notepad?

The answer is "Heck yeah!" Why is this true? Because in reality, a CFC is JUST A TEXT FILE with a '.cfc' extension. It's the content, however, of this text file that makes it special, because what it contains are the blueprints that Coldfusion uses to create living breathing instances of objects. So, when you tell Coldfusion something like "<cfset objUser = createobject("component","model.user")>", you're actually saying "CF, go find the user.cfc file, open it up, read the contents, and give me back a living breathing OBJECT that is built according to the blueprints."

In a very small nutshell then,

A Class is the definition of an object.

The class tells Coldfusion what methods to create, what variables are available, what items are returned...gosh, it tells CF every little detail about how the finished product (object) should behave and respond!

Class = CFC = Blueprint

For contextual illustration, a Java ".class" file is also just a text file containing the blueprints that Java needs in order to construct a Java object. A CFC is a *Coldfusion* class containing the blueprints CF needs in order to construct a *Coldfusion* object.

The "plan"...the "blueprint"...the CFC... it's what the rest of the OO world refers to as a "CLASS", and so should you!

CONSTRUCTOR

A coldfusion "constructor" is any code and or variables that are located outside of a cffunction tag within a CFC, and is typically used to set up variables that will need to be referenced by any function within the CFC. Having variables set up in the constructor allows functions to share values between themselves and between function calls. A very common need for a constructor would be your application's DSN. Many functions within a CFC may need to execute queries, so rather than having to pass in the DSN every time you make a function call, pass it in one time the first time you instantiate the CFC as an object. Consider the following example:

(your component...)

```
<cfcomponent >
```

```
    <!--- this variable will hold a log of progress that can be added to and returned by any
function within this cfc --->
```

```
    <cfset variables.rsslog = "">
```

```
    <!--- this variable will hold the dsn to be used by any function within this cfc --->
```

```
    <cfset variables.dsn = "">
```

```
<cffunction ...
```

```
</component>
```

(our instantiation of that cfc...)

```
<cfset myObject = createobject("component","testcomponent").init(dsn=application.dsn)>
```

Variables set up within the constructor will typically be in either the THIS or the Variables scope. Quite often, as in the example, you will find them being populated at the time an object is instantiated. See INIT METHOD for much more detail on this.

Although technically constructor type code can be anywhere within a CFC (as long as it's not within a cffunction tag), best practice is that they should all be located at the top of the component, before the first cffunction tag. That's how you'll find the CF experts writing them, and it's certainly easier to read when you do it that way.

You should note that it is becoming increasingly popular to not have anything outside of a CFFUNCTION tag and just move all of the CFC's global variables inside an INIT method instead. In that instance, you can (and should) rightly refer to your INIT method as your constructor.

One more thing, while I'm thinking about it...Anytime you read about constructors in relation to CFCs, it is almost always preceded by the prefix "pseudo", as in "pseudo-constructors". I'm quite certain there are very valid reasons for calling ColdFusion constructors pseudo, and there are probably entire theses in existence that could prove that ColdFusion cannot have an authentic constructor to save its life. HOWEVER, every "pseudo" constructor you create will work and do what you want it to, so let's just call them constructors and, since we're using ColdFusion, forget about the fact that Java can make real ones; it keeps it simple.

DAO

When you hear or read the acronym 'DAO', always think "database actions dealing with a single record". DAO is one of the three "Model sisters" (the other two are Gateway and Record (who also holds the nickname "Bean" in some circles)), and is a CFC that contains a collection of basic functions to allow one to create, read, update, and delete a single record in a database table. If your app needs to perform some action on a single record, create a DAO to help out with that task.

FACTORY PATTERN

Factory Pattern...this is another one of those "patterns" that I do not consider to be a pattern of any sort. From my real world experience, an object can BE a Factory, or you could even create an entire application (or framework) whose job it is to BE a Factory...but a pattern of factories? Nah, doesn't gel.

A Factory makes things, right? And that is precisely what is being referred to when someone talks about a Factory, or the Factory Pattern: an object or framework whose job it is to create other objects for you. So, rather than create your own object yourself like so:

```
<cfset myObject = createobject("component",model.myObject)>
```

You would call upon your Factory object to make it for you, similar to the following:

```
<cfset myObject = myFactoryObject.MakeMeAnObject(obj = "model.myObject",args = stArgs)
>
```

If you opt to have an object create your other objects for you, then YOU, O Best Beloved, are a user of the Factory Pattern, and may boast of it in public settings as the mood strikes you.

Here's a question that's BEGGING to be asked: WHY would anybody wanna add that level of complexity to their app???

I'll have to respond to that question with another question: Has it crossed your mind that sometimes, perhaps, one object might need an instance of another object inside of itself in order to do some kind of work?

Whether it has or hasn't crossed your mind, the fact is that many times this will be the case, especially when you're writing your app around an MVC framework. Now, the traditional method of making one object available within another object would be to simply write a line of code within your CFC that instantiates the other object, such as in our example above. However, what if you are using this same object...we'll say, an object that performs emailing duties, within many other objects? Then one day you make a change to the Email.sendMail() method that requires an additional parameter be passed in. You would have to go to every other CFC that instantiates the Email.CFC and modify that line of code to accommodate the new parameter. Could take a long time, you might miss one, etc. Using a Factory, however (such as the Coldspring framework, which I am totally in love with), you can make that change in one place and the Factory will ensure the change is cascaded appropriately.

So what would be a real world scenario where I would actually need one object inside of another? Hmm...how about registering a new user for access to a site? The steps involved with registering a new user are: create the user record, send the user an email. In order to keep my functionality all segregated nice and neat so i can use it here and there, I have myself a User object that handles user records, and an Email object that performs emailing duties. Since my registration process means I need to be able to perform functionality from both the User object AND the email object, I create a third object called RegistrationService.CFC that will do nothing more than orchestrate, or coordinate, work that the User and Email objects know how to do. This scenario is just one of many compelling reasons to utilize a factory.

Oh, and by the way, that coordination of work is what makes RegistrationService a service layer object...not the fact that it has 'Service' in the name.

FRAMEWORK

The result of a lot of mental work figuring out the most efficient way of organizing our code so that it is able to evolve with as little pain as possible. Fusebox is a framework...it allowed us to organize our apps according to functionality and keep that functionality segregated to some degree. My buddy John's self-posting forms are a framework (albeit not a very desirable one); A framework is just the code in place that we leverage to keep our applications neat, tidy, and growable. In an OO context, 'Framework' will refer to those frameworks that are designed to keep *objects* organized within an app and conforming to a few "OO Rules" that span many programming languages.

FUNCTION

The Definition: A function receives something into itself, performs some kind of work on

whatever the input was, and spits the results of that work back out. Period.

It can be a math function that accepts a value 'x', plugs it into some equation, and then returns the mathematical result; it can be a programmatic function that receives something into it...be it a numeric value, a string of text, an object, etc....and returns the results of the code it was told to execute. Ah, and here's the clincher, the missing link that will help ANYBODY, I mean even your TODDLER, to completely understand what a FUNCTION is:

Even your own BODY is a function! You give it some kind of input...let's say, a peanut butter sandwich. Your body performs some kind of WORK on that input (in our case, digestion), and VOILA! You get some output. In this case, a turd.

There is absolutely NO difference whatsoever between a calculus function, a programmatic function, the Human function, or even the myriad of individual functions that work in harmony within our own ecosystem. A function receives something in, performs some work on it, and gives something back.

GATEWAY

When you hear or read the term 'GATEWAY', always think "a CFC whose job it is to return query results". The key thing that sets Gateway apart from its sisters 'DAO' and 'RECORD' is that it almost always deals exclusively with the retrieval of multi record data sets, and not much more than that. If your app needs a query to play with, ask a GATEWAY to get it for you.

GETTER

What you call any CFC method that GETS a value for you. This value could be a query, a structure, a string...it can even be another object. The important point to remember (and actually a difficult one to forget in this case) is that a Getter "gets".

Getters are a "best practice" way to interact with an object's internal variables. For instance, let's say we have a 'User' object. This object is obviously going to have some properties like 'firstname', 'lastname', etc., right? Now let's say that we need to get the value of one of those properties. Our first tendency would be to do something like this:

```
<cfset firstname = UserObject.firstname>, grabbing the property's value directly.
```

But for lots of reasons, this is typically considered the "not so right" way to go about it. Instead, we should give our User cfc a "getFirstname" method, which will return to us the value of an internal firstname variable stored within the object itself and out of the direct reach of any external call. Like this:

```
<cfset firstname = UserObject.getFirstname()>
```

Alright, enough on Getters.

IMPLEMENTATION

Choose the sentence that makes more sense to you:

1. "Hey Scott, would you mind divulging your new method's implementation when it is passed a null argument?"

2. "Hey Scott, what the heck is that method doing when I call it without passing in a value"?

Let's just KISS it, shall we? When you hear or read the word 'Implementation', it is merely referring to HOW a component accomplishes it's tasks. I've never had occasion to use this term, and I'm pretty sure none of The Rest of Us ever will except when talking to people who have adopted it in lieu of more natural phrasing. But, at least now you know how to think about it.

INIT METHOD

A practice inherited from the rest of the OO programming world (and a good practice, I might add), an INIT METHOD is a function within a CFC that is typically called the first time you INITIALize an object using that CFC. Like this:

```
<cfset myNewObject = CreateObject("component","com.rss").init("myDSNname") />
```

Why do this, you might ask? For the same reason that one might use an application.cfm template, or an app_globals.cfm template: It's the place where you can "set up" your CFC with things that will apply to all methods within that CFC (like DSNs). Oh, and one thing that always holds true with an INIT method: it will return THIS. Here is a sample init method:

```
<cffunction access="public" name="init" output="false" returntype="COM.RSS">
    <cfargument name="dsn" type="string" required="yes" default="">
    <cfset variables.dsn = arguments.dsn>
    <cfreturn this>
</cffunction>
```

A few points to note:

1. The returntype attribute of the cffunction tag is set to the actual path and name of the CFC itself. In this example, the CFC is named RSS.cfc, and resides in the mapped directory 'COM'.
2. Any variable residing within the 'Variables' scope within the CFC is visible by any and all methods of that CFC at all times. Definitely a shared scope.
3. Returning THIS will result in the return of an object that contains all of the functions found within the CFC. Create an object with a CFC that returns THIS and then dump it with CFDUMP and examine its contents.

INTERFACE

Think about this: we know what a USER interface is...it's what is presented to the user that allows them to interact with our application. For example, let's give our user a username field, a password field, and a GO button. Those three items compose the User Interface, yeah? Well, an OBJECT's interface is what it will present to our CODE that the code can use to interact with it. Could possibly be an INIT method, a GETLOTTERYNUMBERS method, and a BUYLOTTERYTICKET method. Those three items then would be that object's INTERFACE. So, when you hear the word "INTERFACE", just imagine a mental list of all of the methods that a particular object has.

On a side note...most of the time that I've seen this term used in a ColdFusion context, it's in

sentences similar to this: "ColdFusion doesn't support interfaces". In this context, you are probably looking at some part of a Java/CF debate that really doesn't concern The Rest of Us. From where I'm standing, the only time you need to be able to explain in any detail just exactly why ColdFusion doesn't support Java-type interfaces is when you actually CARE that it doesn't, or you're on "Who Wants to be a Millionaire", and that happens to be the question at hand. I, for one, have yet to have a reason to care that CF lacks this ability and in fact am one of those individuals who couldn't support or dispute this supposed fact if my life depended on it. So, just be aware that there are times when you'll be hearing or seeing this term used in a context other than my initial definition.

METHOD

"Method" = "Function", and "Function" = "Method". When you write a CFC, you use CFFUNCTION tags...to...create...functions. However, when you slice it all down to the nitty gritty, a cffunction tag within a CFC really isn't a function from the perspective of the rest of the OO world; It's a Method. So, be a conformist and just start calling your CFFUNCTIONS that reside within the CFCs that represent your objects METHODS, and your wildest dreams will come true.

Pop Quiz!

1. What is the name of the following method: `<cffunction name="init" access="public" output="false" returntype="myCFC">....`
2. What method is being called on the RSS object in this example: `<cfset newval = RSS.GetFeeds() />`

MUTATOR

Anytime you hear someone use this term, just laugh to yourself and say, "Oh, you mean THE SETTER"?

MVC

Ah, there's that little acronym that is in the process of changing your life right now, eh? It stands for Model(M), View(V), Controller(C), and has been around for longer than many of you have been alive, only recently having found relevancy with regard to ColdFusion. MVC is the name of a programming 'pattern', or way of laying out an application's code. The relevant portion of MVC's definition is that it is a way for us to build an application and yet be able to separate its parts in a very organized manner. It does something along the same lines as what fusebox did for us, only in a much more disciplined and cross-industry standardized manner (MVC is a common term among programmers of many different languages, so once you understand it for one, it's the same for all the others). With the different areas of our application segregated using the MVC pattern, we can give each part the attention it deserves without affecting or being distracted by the other parts of it. Let's look at the three parts of MVC individually...

VIEW

View is the V in 'MVC'! (dang, that rhymed). What's a view? Yes, the obvious is correct: it's what the end user will see with his or her eyes. They'll be "viewing" it. Get it? Get it? To be a bit more specific about it, 'view' represents that portion of an application's code that is responsible for creating the output a user will interact with...forms, text, layout, etc.; the user interface. When

you think 'view', think user interface.

CONTROLLER

The Controller, in a nutshell, is the liaison between your View and your Model. For instance, the View needs certain variables to be present and accounted for in order to output its display, right? Well, the Controller's job is to make the necessary calls to the Model (who will typically supply the values and perform business logic), ensuring that the View's expectations and needs are met. Let's think about an MVC-type application, for instance, and it's time for the user to log in. Okay, they enter their username and password, hit enter, and BANG! They're off! The user info is submitted to the app; the app's framework (Model Glue, Fusebox, Mach II, whatever) stuffs those values into a bucket (called 'Event'), then broadcasts a message called 'LOGIN'. The Controller is actually waiting for and listening for a message called 'LOGIN', and when it hears it, it knows that it can be certain there are user credentials sitting in the EVENT bucket. Controller's LOGIN method is called. The LOGIN method of the controller makes a call to a component in the MODEL that performs security checks, passing it the username and password. The MODEL responds to the Controller with a Success or Failure notice, and if Success, also the user's information and permissions. The Controller takes the information returned and puts it into the EVENT bucket. At that point the View is called, and the view template displays the values it finds waiting for it in the Event bucket. Besides being Liaison, the Controller is also the only one who is allowed to talk to the SESSION and APPLICATION scopes. Although Model and View are 'capable' of doing it, it's a violation of MVC for anybody but Controller to do so. Just things to keep in mind when deciding what functionality to put where in your MVC app.

MODEL

Model. Now there's a term that The Rest of Us have never had the need to use other than when talking about the Swimsuit Edition of Sports Illustrated or our model car collection. In OO, it is one word that can mean two different, yet related things, but context will usually keep its meaning clear. Model is the M in 'MVC', and so from this day forward, whenever you speak about that portion of an OO application's code that is responsible for interacting with the backend database and/or executing business logic (such as verifying user credentials), you will refer to that portion of code as "The Model". IN ADDITION, from this day forward you will ALSO begin to refer to your application's backend database as "the application's model". Please repeat the following vocabulary sentences aloud to help drive these definitions home:

1. "The database supporting my application is my application's data model".
2. "Which one of you guys is going to handle designing the model portion of this application?"
3. "No sir, I'm not sure just where that variable's value is being stored. Let me go check the model...."

OBJECT

Making a distinction between the definition of an Object versus a Class is probably not a vital factor in one's OOP abilities. I believe, however, that if we're going to have and use terms, we should do so accurately in order to effectively communicate. I absolutely hate it when people misuse words in ignorance, such as when someone refers to a Cicada as a Locust (a locust is a grasshopper! a Cicada is a Cicada!), or a bat as a rodent (they're both mammals, but that's as close as it gets). For clarification's sake, then, following is what I have discovered to be the definition of an Object in a Coldfusion context.

When asked the question "what IS an object?", I nearly always respond with a description that

includes the phrase "living and breathing". Here's why:

According to science, the basic checklist to use when determining if something is alive or not is:

- Can it Reproduce?
- Can it Obtain and use energy?
- Can it Grow, develop, and die?
- Can it Respond to the environment?

Once you issue the command

```
<CFSET objFrankenObject = CreateObject("component","model.Frankenstein") />
```

Coldfusion takes what is a lifeless blueprint (the Frankenstein.cfc file) and breathes life into it, creating something that meets every one of these criteria!

- Can this Creation reproduce? It certainly does have the potential, depending on how it was designed.
- Can it grow, develop, and die? Abso-frickin-lutely! It occupies a certain amount of ram, and I guarantee you that as soon as I call one of its setters and stuff a large structure into it that it will grow and then occupy MORE ram! If I `<CFSET objFrankenObject = ""`, it'll be dead and gone.
- Can it respond to its environment? Of course; that is what it was created to do in the first place.
- Finally, can it use and obtain energy? Without spending too much effort in attempting to stretch the analogy into that realm...let's just say 'yes' since our server is plugged into an active outlet.

By all accounts, this Object is a living, breathing, grammatical entity. This way of thinking about objects in contrast to the flat, lifeless CFC/Class (that so often is referred to as an object, erroneously in my opinion) makes the distinction between them crystal clear.

In a nutshell then:

Object = living breathing instance of a CFC (Class)

Again, thinking of objects in this way may not make one a better or worse OO programmer, but it definitely doesn't hurt to have a finer understanding of what the terms truly represent, right?

OO (pronounced "Oh! Oh!", and with just as much zeal!)

Object Oriented. The same way a person has a sexual orientation, so do applications have an orientation. An "object oriented" application has the tendency toward utilizing a butt-load of objects and a suitable framework for keeping them organized. OO with regard to ColdFusion means that our butt-load of objects are CF Components that will be used almost exclusively via a `CFOBJECT` or `CREATEOBJECT` call, and rarely if ever via `cfinvoke`. OO with CF also means that we have enlisted the help of a suitable "OO" framework (like Model-Glue or Mach II) to help us keep it all organized.

PATTERN

Nothing magical, really, about this term. My grandma made patterns in her quilts, wind produces patterns in sand dunes, and CF coders who use OO methodologies (or don't) end up having patterns within their codebase...a consistent, characteristic form, style, or method of doing something. The number of different OO patterns that exist are quite numerous, in fact there are voluminous books written that describe them in detail. Each has its own common name, each has its own purpose and thing that it's good at, and rarely is any pattern used exclusively within an OO app. For instance, the Singleton pattern is good at making sure an object only gets created one time within your app, and not multiple times for multiple users; The Memento pattern is good at making sure you're always able to roll back changes the user decided they didn't really want to make; etc. The list goes on and on. Bottom line: when you hear someone talking about an OO pattern, start listening for it's name and what its specialty is. Believe it or not, you too will one day be referring to these little guys when talking about the OO apps you've written!

Vocabulary Sentences:

1. "D'OH! I see why your value is being overwritten! The particular controller you're calling was instantiated as a Singleton!"
2. "Good suggestion, Sven. I agree that we should instantiate those controllers using a Factory."

RECORD

See BEAN

RETURN TYPE

This term is in my lexicon because now that you're entering the CF OO world, you are going to be returning "custom types". This is referring to the 'returntype' attribute of the CFFUNCTION tag. In the procedural world, I never once had an occasion to return anything other than those common types like STRING, QUERY, ARRAY. However, when using CFCs in an OO context, those little suckers are going to very often be returning THEMSELVES as fully formed, functioning, living objects. When it is the case that a CFC will be returning itself, the ReturnType will be the full cfc path to the component, such as "cfc.controllers.mycontroller". ah, and be sure to omit that final ".cfc" extension...it doesn't like, want, or need it. Here's an example of what I'm talking about:

```
<cffunction access="public" name="init" output="false" returntype="com.RSS">
  <cfargument name="dsn" type="string" required="yes" default="">
  <cfset variables.dsn = arguments.dsn>
  <cfreturn this>
</cffunction>
```

Returning a type of "com.RSS" means that the item being returned to the caller (in this case, an instance of the RSS.cfc component itself, which lives within a mapping named "com") must match in every way the structure of the com.rss component. Sounds redundant, I know, because how could an object not be identical to itself? But consider this possibility: It is possible for me to

create a CFC that returns itself, but make the Return Type equal to a totally different component AS LONG AS the two components have the same methods, arguments, and constructor variables. There won't be many occasions to do this probably, BUT it does illustrate the fact that using Return Type to refer to an actual component is to ensure that what is being returned really does match up exactly with the definition of the component referred to in Return Type. Geez, hope this makes sense.

Suffice it all to say, that when returning an instance of a component using a <CFRETURN THIS> tag, make sure the Return Type specified is the complete path to the CFC itself. Otherwise, you'll get the ol' "Object is not of type bla bla bla" error.

SERVICE LAYER

This is not NEARLY as gray and ambiguous a term as you might think. Picture if you will, a man sitting comfortably on his sofa. In one hand is the remote for his very large plasma TV; in the other hand is a remote for his home theater system. The two remotes and the man are all objects, and all three come pre-built with things they can do. In the kitchen is the man's wife; let's think of her as the calling application. She barks out the order to the husband object, "START THE MOVIE, YOU IMBECILE!". The husband object just happens to have a startTheMovie method, and begins to execute it. First, he manipulates the objTVRemote object, calling its "tvOn" method. Then, he manipulates the objDVDRemote object, calling its "dvdOn" and "dvdPlay" methods. Now he manipulates the objTVRemote again, calling the "inputSRC" method and switching the tv to receive the dvd input. Tada! Movie is playing now!

Pretty clear scenario, eh? Well, in this illustration, the MAN is acting as the SERVICE LAYER. Although he has a "startTheMovie" method, all he's really doing is coordinating efforts between other lower level objects that actually do the work. His wife doesn't care about the remotes or how they work, and her life is then simplified because she need only make her one call to her SERVICE LAYER OBJECT and he handles the nitty gritty details. Service Layer...not such a deep, complicated mystery after all, is it?

SETTER

Opposite of a Getter, a Setter...does what? That's right! It SETS values! Tell him what he's won, Johnny! And by the way, everything written in the definition of "Getter" completely applies to Setters as well.

SINGLETON

You, O Best Beloved, are a Singleton. There is nobody else like you in the whole wide world, nor has there ever been. You're a singular occurrence, a unique random combination of genetics that exists only wherever you happen to be at a given moment. And if anybody wants to interact with you, they know just where to find you. You can interact with lots of different people, but no matter how many friends you may have, every one of those friends is interacting with the exact same YOU, and not a clone. When your good friend Dave asks you to tell Theresa hello next time you see her, you store that information and next time you interact with Theresa you pass on that exact message.

You getting the picture here? If you instantiate a CFC and put it in a place (Application scope, anyone?) where ANY other part of your application is able to interact with THAT

PARTICULAR instance of your CFC, then YOU, O Best Beloved, have created your object as a Singleton. Now, for whatever reason, people seem to be awfully fond of tossing around the buzz phrase "Singleton Pattern"...not sure why. It's way simpler just to use the word as an adjective describing how you have instantiated your object, and it makes more sense, too.

Hmmm... a few other thoughts on Singletons...

It's vital to understand this term because it has a huge effect on how your app will work...basically, if an object is created as a Singleton and you aren't able to visualize what that means, you could have people seeing other people's data. Not good.

That I'm aware of, there is no opposite of Singleton...an object either IS a Singleton, or it isn't.

POP QUIZ!

Which of the lines of code below is creating my object as a Singleton?

A) <cfset adminObject = createobject("component",model.admin) >

B) <cfset application.adminObject = createobject("component",model.admin) >

THIS, VARIABLES, and VAR Scopes

These are three of the variable scopes that are found within the world of a ColdFusion object, and three scopes that can make you pull the rest of your hair out when you don't know how to think about them. In a nutshell, you're looking at three increasing levels of variable privacy, from most liberal to most private.

The "THIS" scope holds items that can be directly accessed from anywhere inside OR outside of the object itself. Consider the following example of an instantiation of the myTest.cfc that has a variable called THIS.GLOBALVAL within it's INIT method:

```
<cfscript>
    myTestObj = CreateObject("component","myTest").Init();
    myTestObj.GlobalVal = "I set you from outside of the object!";
</cfscript>
```

In this example, because the variable GlobalVal was put into the THIS scope within our object, our application could directly access it as a property. Cool, if that's what you intended to happen. Not cool if it wasn't.

The VARIABLES scope within a component object is a scope that can be accessed by any method within the object at any time, in real time. In other words, if our component had set up a variable called Variables.LimitedVal, all methods will be sharing that one instance of the variable. If method one sets it to "5", and later the app calls method two which reads that variable, it will see the value "5". Any attempt, however, from outside the object itself to manipulate that value will result in an error. The following would FAIL:

```
<cfscript>
    myTestObj = CreateObject("component","myTest").Init();
    myTestObj.LimitedVal = "I set you from outside of the object!";
```

</cfscript>

And finally, the VAR scope. This scope is one which can be seen only from inside of the actual method itself. For example, I can have three methods, each that use a variable with the same name that was initialized within the VAR scope, and no method will ever see the variable used by the other methods. It is a VITAL thing that you initialize your private variables in the VAR scope inside of your methods, because by default they are set up in the VARIABLES scope, and who knows WHAT havoc will occur if you have methods sharing variables that were intended to be private. Consider the following sample of initializing a variable in the VAR scope:

```
<cffunction access="public" name="sampleMethod" output="false" returntype="void">
    <cfargument name="headlines" type="array" required="yes" >
    <cfargument name="sourceID" type="numeric" required="yes">

    <cfset var iterations = 0>
    <cfset var urlitems = "">
    <cfset var newInsertItems = arraynew(1)>
</cffunction>
```

Only the 'sampleMethod' method will be able to see and manipulate those variables set using the 'var' scope.

As an aside, I have recently been informed that 'VAR' isn't actually a scope, but rather a special sub-scope of the VARIABLES scope. Good info...but for all intents and purposes, you can and should think of VAR as a scope of its own.

TRANSFER OBJECT (aka 'TO')

Can you say "unk mangani" (that's what tarzan used to say to the elephants)? A Transfer Object is a close relative of the Bean, only much more primitive. In fact, if it weren't an object, it would be a structure because all of its values are accessible in a very direct way without the need to call a 'get' or 'set' method. For instance, to get the first name from a user transfer object, you would simply say "<cfset firstname = myUserTO.firstname>". Transfer Objects, from what I have seen, are very well suited for "transferring" static data between objects and are basically meant to be disposable and not persisted. As an aside though, using TO's isn't mandatory or even necessarily a best or worst practice; they are simply an option for you to consider, and since Reactor generates them for you automagically in MG Unity, they are there if you want to use them.

MG TERMS

[Back to Top](#)

BROADCAST

Broadcast is the Model-Glue 'Towne Crier'. He is sent into action by the specific event he is waiting for, and as soon as it happens he lives up to his name and broadcasts. And what, my best beloved, do you think Mr. Broadcast actually broadcasts? Yes! Messages!

CONTROLLER

This item was already defined in the OO section of the Lexicon, but it merits additional attention in this section as well. The same definition as was given in the OO section still applies, with the added caveat that in Model-Glue, the Controller is really a singular way of referring to your application's *collection* of controllers, because most of the time there will be several discreet controller CFCs present in your MVC app. For instance, if two developers are working on a single MG application, and one were to say to the other, "Hey, can you check the code on method X in the controller?". The other developer will definitely respond, "Uh, dude, which controller are you talking about?".

Now, to continue with the story of our ill-fated Message Bud and his faithful horse, it was the Controllers that he was visiting when he got sent out by Broadcast. He went to every Controller on the stage, presenting himself to see if any of them were waiting for him to arrive. Now, how is it that a Controller can be "waiting" for Bud the Message? Because a Controller comes equipped with an infinite number of ears, each ear tuned specifically to listen for the arrival of a specific Message. As soon as a Controller ear detects the presence of the Message it was waiting for, it receives any incoming name-value pairs, notifies its Controller that Message Bud arrived, and thus kicks off a very specific piece of work, whatever that might be.

EVENT

As with the term 'Model' that we covered previously, 'Event' can take on two similar yet distinct definitions in an MG app: one as an object, the other in a role almost exactly like that of a fuseaction.

1. Event as an object is the star of the Model-Glue show, without a doubt. When you think of the event object, think of a bucket, because that is exactly how it behaves; things get put in, and things get taken out. The event is, in a sense, alpha and omega in the model-glu world, because it is the first thing created and the last thing to be tossed away, and it is handed off from one thing to another like a baton in a relay race until there are no more things for it to be handed off to. At that point, it has reached the end of its brief lifespan and simply goes away.

LISTENER

Listeners, then, are the ears of the Controllers. Not too much to say about them, except they're listening for a very specific Message, and when they hear it, they notify their Controller to perform whatever piece of work the Listener was in charge of triggering.

A Controller can have as many Listeners as you need, and many Listeners can be listening for the same Message, too.

MESSAGE

Message's job is that of messenger between the different players. When Broadcast is tagged by Event, he immediately sends out whatever Messages he has been assigned to oversee. Every Message has a name, and the ability to carry additional extraneous values when needed. For example, picture a Pony Express Rider named Bud waiting in the stable of a mail center for the SENDMAIL event. As soon as the SENDMAIL event is received (via telegram, of course), the Broadcast runs out to the stable and slaps Bud's horse on the rump roast. Message Bud immediately runs out, presenting himself to every other player who might possibly be interested. Oh, and on Bud's saddle is a saddle bag with two or three name-value pairs. Whoever he presents himself to who was actually waiting for him to arrive, Bud hands them a copy of the name-value pairs and tells them to get to work. Once Bud has visited every player who might be interested, Bud and his horse promptly die. Sad, I know, but at least he lived a purposeful life. In fact, Bud's life and job are so meaningful that he alone single-handedly gave weight and meaning to the term "Implicit Invocation", because Bud is able to deliver without even having a good address to deliver to. Now isn't that special?

Bud's customers are the next players in this Model-Glue drama that we'll get to know. They are called The Controllers.

MODEL

Same definition as was given in the OO section, but with the addition of the fact that when working in and talking about Model-Glue, most of the time a reference to the Model will be referring to the CODE used to talk to the database, and not the actual tables themselves.

Model-Glue:Unity

Model-Glue: Unity comes bundled with two other frameworks: Coldspring, and Reactor. They're all 'united' into one composite framework, thus the name 'Unity'.

MODELGLUE.XML

This is the majority of the heart and soul of any model-glue application; it is what punchcards were to the Eniac, what DNA is to living organisms; it defines what happens, when it happens, and how different events interact with one another. It is the blueprint for programmatic flow, man. It just doesn't get any meatier than this!

As the file name implies (quite loudly), this is an XML file. The Model-Glue framework itself relies completely on this file to perform the magic of "Implicit Invocation", and does so by utilizing a set of tags created specifically for this purpose. There are many tags and tag attributes available for use within ModelGlue.xml, but let's look at the core tags at a high level.

CONTROLLER

We know what a controller is from the previous definition of it. Well, this tag is where you define what controllers are available to the app, and what messages each controller will be listening for. Like so:

```
<controllers>
    <controller name="AuthenticationController"
type="controller.AuthenticationController">
```

```

        <message-listener message="OnRequestStart" function="SetCurrentUser"
/>
        <message-listener message="OnRequestStart"
function="CheckCurrentUser" />
        <message-listener message="UserNeedsPermission"
function="CheckUserPermission" />
        <message-listener message="NewUser" function="createUser" />
        <message-listener message="Logout" function="logoutUser" />
        <message-listener message="Login" function="authenticateUser" />
        <message-listener message="makeKeychain" function="makeKeychain"
/>
    </controller>
</controllers>

```

What we name our controller isn't all that relevant and is more for readability than anything; but the TYPE we specify, that must contain a valid path to an actual CFC. Each message-listener tag's message value is the name of a broadcast that will trigger the function named in the 'function' attribute. Pretty simple, eh? Whenever one of these functions gets triggered, the Event bucket we talked about before is automatically available to it. OH, one more cool thing. Model-Glue has some "built in" events that your controllers can listen for. The most commonly used ones are OnRequestStart and OnRequestEnd. As their name implies, these messages are broadcast at the beginning and end of each request, and come in very handy for doing the things that at one time you may have employed your application.cfm or onRequestEnd.cfm templates for.

<EVENT-HANDLERS> <EVENT-HANDLER>....

Now HERE is where actual events are configured. The equivalent of your index.cfm page in a fusebox 2 app, or an FBX_SWITCH file in a fusebox 3 app, all of the possible named events (again, similar to a fuseaction in functionality) are defined here. There are three child tags of the <event-handler> tag: <broadcasts>, <views>, and <results>. Here's a robust sample event:

```

<event-handler name="page.landing">
    <broadcasts>
        <message name="makeOrgTree" >
            <argument name="format" value="STRING" />
        </message>
        <message name="getUserToDoList" />
        <message name="getLinks">
            <argument name="section" value="all" />
        </message>
    </broadcasts>
</event-handler>

```

```
</broadcasts>
<results>
  <result name="LoginNeeded" do="home" redirect="true" />
  <result do="view.template" />
</results>
<views>
  <include name="todoList" template="dspToDoList.cfm" />
  <include name="userInfo" template="dspUserInfo.cfm" />
  <include name="orgDropdown"
template="frmSelectOrganization.cfm" />
  <include name="currentSelectedOrg"
template="dspCurrentSelectedOrg.cfm" />
  <include name="body" template="dspLanding.cfm" />
</views>
</event-handler>
```

BROADCASTS

As you can see, some of the child tags have children of their own, enabling you to configure your event in great detail. Walking through the event above, Model-Glue will first make all of the broadcasts specified. "makeOrgTree" is sent out first along with an additional argument value for use by the method being kicked off. Each of the broadcasts are performing some piece of individual work, and probably also adding something to the event bucket (and viewstate) along the way that will be later utilized by individual views in the viewstack.

VIEWS

The VIEWS section tells Model-Glue which templates to render, with the final rendered view being the only one that will get displayed. Consequently, it is important to remember that each of the previously rendered views will be included within the final template, almost exactly the same way you would do a <cfinclude>.

RESULTS

Ah, and lastly we have the Results. These exist for the purpose of performing the post-event activity of forwarding the user to another event. Results are named (as the 'name' attribute implies), and are set via code within a controller. For instance, if we were performing a login event, we could have the login controller set a named result of either "success" or "failure" so that our login event would know which way to direct the user at the end of that event (either to a 'sorry, login failed' page, or to the landing page for an authenticated user). Notice also that the <result> tag can have an attribute called "redirect". Even though any named result will automatically take the user to the event specified in the "do" attribute, setting redirect to "true" tells Model-Glue NOT to preserve the current event bucket, but to start a new, empty one. Also notice that the results doesn't have to have a name; in this case, the result will be executed without regard to whether or not that named result exists...it's just gonna do it no matter what.

VIEWSTACK

Oh man, just picture a large stack of flapjacks! One pannycake, two pannycakes, three pannycakes...can you see them? All stacked up like that? The VIEWSTACK is a stack as well, just like that, only instead of pancakes we're looking at a stack of view templates that have already been rendered by CF into pure HTML. For instance, let's say that for a certain event called 'LOGIN' (there's that LOGIN again!), we have our resulting display page broken into several different pieces. We have a main body of content (dsp_content.cfm), we have a login area that either displays the login form or a message saying who is logged in (dsp_login.cfm), a footer section (dsp_footer.cfm), a header section (dsp_header.cfm), and then finally one template to arrange all the pieces (dsp_layout.cfm). Okay, during the VIEW portion of the LOGIN event (fuseaction), Coldfusion is directed to first render dsp_login.cfm, then dsp_footer.cfm, then dsp_content.cfm, then dsp_header.cfm, and finally dsp_layout.cfm. Each one is rendered as an individual item, and as soon as CF finishes with it, it adds the rendered HTML to the VIEWSTACK (giving it whatever name the programmer said it should), just like tossing another flapjack onto what's already waiting there. Now, the order in which the individual view templates are rendered is completely irrelevant, EXCEPT for the fact that it will always be assumed that the FINAL display template rendered is the one that will be laying out all the other ones. That's why in this example dsp_layout.cfm was done last, because within dsp_layout, all of the other parts will be output.

VIEWSTATE

You can think of the VIEWSTATE as the collection of variables and other miscellaneous items that are available to any VIEW template. Whereas the EVENT bucket is the container for all things available to the Controllers and Model CFCs, the VIEWSTATE is there for the VIEW. Very often you will find the VIEWSTATE containing nearly identical content to the EVENT bucket.