“Say What?” Handling Unknown Messages with onMissingMethod() by Sean Corfield

Among the new functions introduced in ColdFusion 8, onMissingMethod() is one of the more unappreciated ones. Most people see and use it as an error handler, if they use it at all. But the function really has so much more to offer. When it came to finding an author for this article, we didn’t have to go far. We didn’t have to go anywhere, actually. Sean Corfield eagerly volunteered before we could even ask him.

If you are using ColdFusion Components (CFCs), you probably think in terms of calling methods on objects. A CFC has a collection of methods defined within it using the \texttt{cffunction} tag, and those declared with \texttt{access=“public”} can be called by other code. For the purposes of this article, there is another way to look at methods and method calls—as messages sent to objects—which I think will help you write better object-oriented code.

Get the Message?

The literature on object-oriented programming (OOP) sometimes refers to the concept of objects passing messages to each other. Instead of describing the objects as interacting by calling methods on each other, some books and articles describe a series of autonomous objects that pass messages around back and forth—messages that request that actions be done (method calls) and messages containing information (return values).

Thinking of object-oriented systems in this manner helps us focus on the interactions within the system rather than on calling a series of functions, which is inherently procedural. Some OOP languages use this message-passing idiom more than others. Smalltalk and Objective-C are classic examples of languages where what we might think of as methods on objects are described, instead, as the behavior triggered when a message is received—and the language syntax reflects this idiom. Listing 5-1 is a small example of Objective-C code.
Listing 5-1. An Example of Objective-C Code

```
// WebHistory Notification Messages
- (void)historyDidAddItems:(NSNotification *)aNotification
{
    // Get the new history items
    NSArray *items = [[aNotification userInfo] objectForKey:WebHistoryItemsKey];
    NSEnumerator *enumerator = [items objectEnumerator];
    id historyItem;

    // For each item, create a menu item with the history item as the represented object
    while (historyItem = [enumerator nextObject])
    {
        [self addNewMenuItemForHistoryItem:historyItem];
    }
}
```

The bracketed expressions in Listing 5-1 represent messages being sent to objects. The following expression means “Send the message userInfo to the object aNotification”:

```
[aNotification userInfo]
```

We would write that in ColdFusion as follows:

```
aNotification.userInfo();
```

When a message contains data, Objective-C uses colons to introduce the data:

```
[self addNewMenuItemForHistoryItem:historyItem];
```

In Listing 5-1 you see the behavior for the message historyDidAddItems, which contains the data aNotification (which has type NSNotification *). Let’s look at another message declaration:

```
- (void)addObserver:(id)observer selector:(SEL)aSelector name:(NSString *)notificationName object:(id)anObject;
```

This message contains four pieces of data, and the message name is essentially broken up into four parts. The full message name is addObserver selector name object, and it would be sent like this:

```
[nc addObserver:obs selector:s name:n object:obj]
```

Looking back at the message declaration, we can see it has four named arguments: observer (of type id), aSelector (of type SEL), notificationName (of type NSString *), and anObject (of type id). An approximate ColdFusion equivalent is shown in Listing 5-2.