CHAPTER 9

Using ColdFusion

In this chapter

Working with Templates

Using Functions

Using Variables

Working with Expressions

Using ColdFusion Data Types

Commenting Your Code

WORKING WITH TEMPLATES

Back in Chapter 5, "Previewing ColdFusion," you created several applications (both using wizards and writing code manually). ColdFusion applications are made up of one or more files—files with a `.CFM` extension. These files often are referred to as *templates*; you'll see the terms *templates*, *files*, and even *pages* used somewhat interchangeably—just so you know, they all refer to the same thing. (I'll explain why the term *templates* is used in a moment.)

CREATING TEMPLATES

As already explained, ColdFusion templates are plain text files. These files can be created with several types of programs:

- ColdFusion Studio (introduced in Chapter 8, "Introduction to ColdFusion Studio")
- Macromedia HomeSite (the most popular code-based HTML and Web editor)
- Macromedia UltraDev (introduced in Chapter 16, "Using Macromedia Dreamweaver UltraDev with ColdFusion")
- Windows Notepad

Obviously, the best choice for ColdFusion developers, as already seen, is ColdFusion Studio. So that's what you'll use here (and throughout the rest of this book).

To create a new ColdFusion file (or template; as I said, the terms are used interchangeably), simply open ColdFusion Studio. The editor will be ready for you to start typing code, and what you save is the ColdFusion file (as long as you save it with a `.CFM` extension, that is).

The code shown in Listing 9.1 is the contents of a simple ColdFusion file named `hello1.cfm`. Actually, at this point no ColdFusion code exists in the listing—it is all straight HTML and text, but we'll get to that soon. Launch ColdFusion Studio (if it is not already open), and type the code as shown in Listing 9.1.

LISTING 9.1 `hello1.cfm`

```
<HTML>
<HEAD>
   <TITLE>Hello 1</TITLE>
</HEAD>

<BODY>

Hello, and welcome to ColdFusion!

</BODY>
</HTML>
```

Note

Tag case is not important, so `<BODY>` or `<body>` or `<Body>` can be used—it's your choice.

SAVING TEMPLATES

Before ColdFusion can process pages, they must be saved onto the ColdFusion server. If you are developing against a local server (ColdFusion running on your own computer) then you can save the files locally; if you are developing against a remote server then you must save your code on that server.

Where you save your code is extremely important—the URL used to access the page is based on where files are saved (and how directories and paths are configured on the server).

All the files you create throughout this book will go in directories beneath the `ows` directory under the Web root (as discussed in Chapter 5). To save the code you just typed (Listing 9.1), create a new directory under `ows` and then save the code as `hello1.cfm`. To save the file, either select Save from the File menu or click the Save button in the Studio toolbar.

Tip

Forgotten how to create directories in ColdFusion Studio? Here's a reminder: Select the directory in which the new directory is to be created, right-click in the file pane below, and select Create Folder.

Of course, you can also create directories using Windows Explorer (or any other application), but if you do so while Studio is open, you'll need to refresh the list in Studio so it sees the new directory.

EXECUTING TEMPLATES

Now, let's test the code. Open your Web browser and go to this URL:

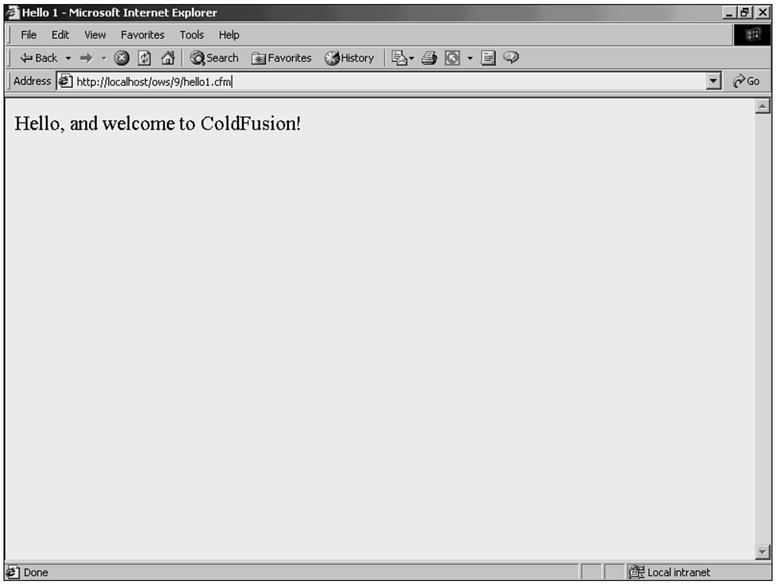
```
http://localhost/ows/9/hello1.cfm
```

You should see a page similar to the one shown in Figure 9.1. Okay, so I admit that this is somewhat anticlimactic, but wait; it'll get better soon enough.

There's another way to browse the code you write. Assuming it is a page that can be executed directly (meaning it is not one that needs to be processed after another page—for example, a page that expects to be processed after a form is submitted), you can browse it directly in Studio by selecting the Browse tab above the editor window (see Figure 9.2).

Tip

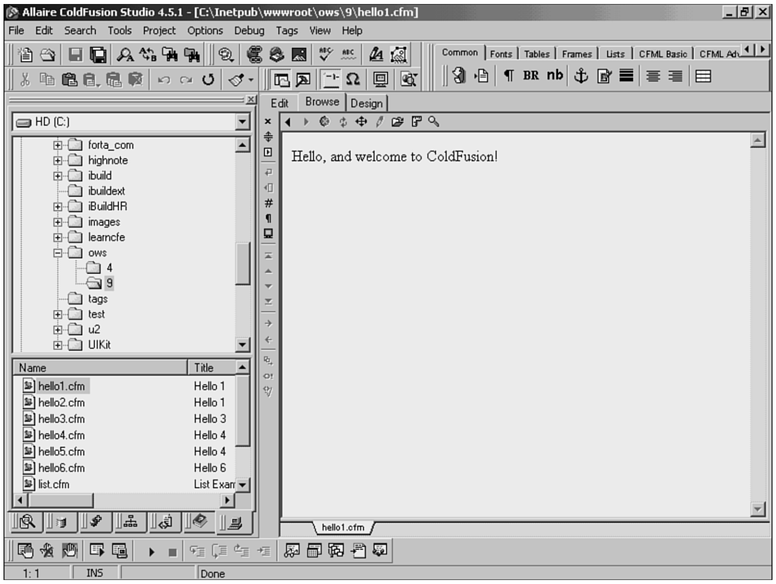
For the Browse tab to work, ColdFusion Studio must have Development Mappings defined. If you run into problems using the Browse tab, the first thing to check is that these mappings exist (Alt+M usually displays the Remote Development Mappings dialog box). Mappings, and how to define them, is explained in Chapter 8; refer to that chapter if necessary.



Hello, and welcome to ColdFusion!

Hello 1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/ows/9/hello1.cfm
Done
Local intranet

Figure 9.1
ColdFusion-generated output usually is viewed in any Web browser.



Hello, and welcome to ColdFusion!

Allaire ColdFusion Studio 4.5.1 - [C:\Inetpub\wwwroot\ows\9\hello1.cfm]
File Edit Search Tools Project Options Debug Tags View Help
Edit Browse Design
HD (C:)
hello1.cfm

Name	Title
hello1.cfm	Hello 1
hello2.cfm	Hello 1
hello3.cfm	Hello 3
hello4.cfm	Hello 4
hello5.cfm	Hello 4
hello6.cfm	Hello 6
list.cfm	List Exam

Figure 9.2
If configured correctly, you'll be able to browse much of your ColdFusion code within Studio itself.

Templates Explained

I promised to explain why ColdFusion files are often referred to as templates. Chapter 1, "Introducing ColdFusion," explains that ColdFusion pages are processed differently from Web pages—when requested, Web pages are sent to the client (the browser) as is, whereas ColdFusion files are processed and the generated results are returned to the client instead.

In other words, ColdFusion files are never sent to the client, but what they create is. And depending on what a ColdFusion file contains, it likely will generate multiple different outputs all from that same single `.CFM` file. And thus the term *template*.

USING FUNCTIONS

And this is where it starts to get interesting. CFML (the ColdFusion Markup Language) is made up of two primary language elements:

- **Tags**—Perform operations, such as accessing a database, evaluating a condition, and flagging text for processing.
- **Functions**—Return (and possibly process) data and do things such as getting the current date and time, converting text to uppercase, and rounding a number to its nearest integer.

Writing ColdFusion code requires the use of both tags and functions. The best way to understand this is to see it in action. Listing 9.2 contains a revised hello page. Type the code in a new page, and save it as `hello2.cfm` in the `ows/9` directory.

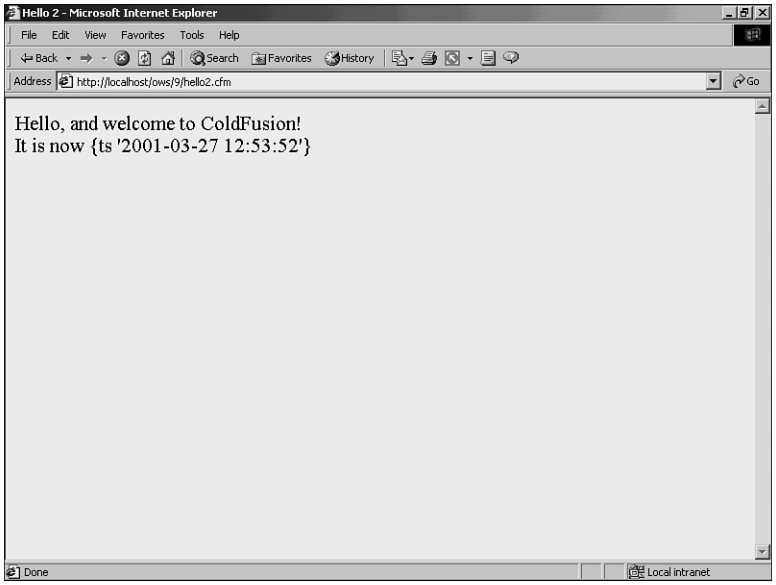
LISTING 9.2 `hello2.cfm`

```
<HTML>
<HEAD>
   <TITLE>Hello 2</TITLE>
</HEAD>

<BODY>

Hello, and welcome to ColdFusion!
<BR>
<CFOUTPUT>
It is now #Now()#
</CFOUTPUT>

</BODY>
</HTML>
```

After you have saved the page, try it by browsing it (either in a Web browser or right within Studio). The URL will be `http://localhost/ows/9/hello2.cfm`. The output should look similar to Figure 9.3 (of course, your date and time will probably be different).

Now, before we go any further, let's take a look at the code in Listing 9.3. You will recall that when ColdFusion processes a `.CFM` file, it looks for CFML code to be processed and returns any other code to the client as is. So, the first line of code is

```
<HTML>
```



Hello, and welcome to ColdFusion!
It is now {ts '2001-03-27 12:53:52'}

Hello 2 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello2.cfm
Go
Done
Local intranet

Figure 9.3 ColdFusion code can contain functions, including one that returns the current date and time.

That is not CFML code—it's plain HTML. Therefore, ColdFusion ignores it and sends it on its way (to the client browser). The next few lines are also HTML code:

```
   <TITLE>Hello 2</TITLE>
</HEAD>

<BODY>

Hello, and welcome to ColdFusion!
<BR>
```

No ColdFusion language elements exist there, so ColdFusion ignores the code and sends it to the client as is.

But the next three lines of code are not HTML:

```
<CFOUTPUT>
It is now #Now()#
</CFOUTPUT>
```

`<CFOUTPUT>` is a ColdFusion tag (all ColdFusion tags begin with `CF`). `<CFOUTPUT>` is used to mark a block of code to be processed by ColdFusion. All text between the `<CFOUTPUT>` and `</CFOUTPUT>` tags is scanned, character by character, and any special instructions within that block are processed.

In the example, the following line was between the `<CFOUTPUT>` and `</CFOUTPUT>` tags:

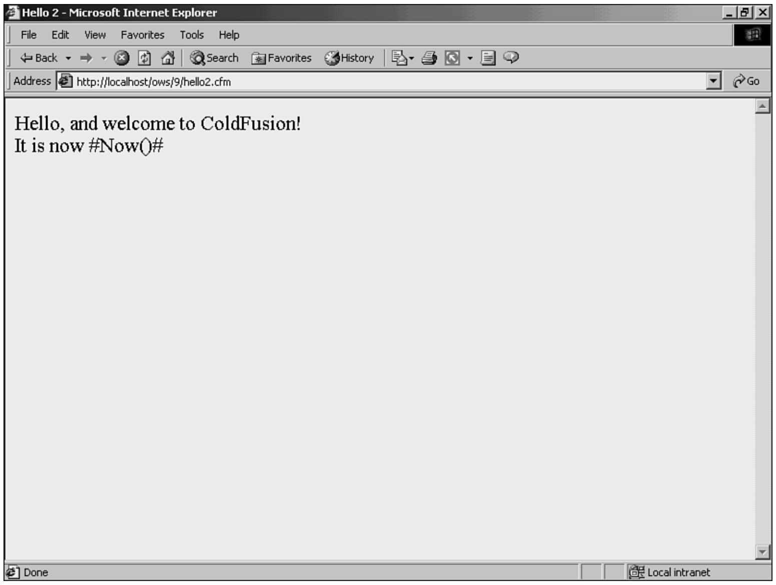
```
It is now #Now()#
```

The text `It is now` is not an instruction, so it is sent to the client as is. But the text `#Now()#` *is* a ColdFusion instruction—within strings of text instructions are delimited by pound signs (the `#` character). `#Now()#` is an instruction telling ColdFusion to execute a function named `Now()`—a function that returns the current date and time. Thus the output in Figure 9.3 is generated.

The entire block of text from `<CFOUTPUT>` until `</CFOUTPUT>` is referred to as a *`<CFOUTPUT>` block*. Not all the text in a `<CFOUTPUT>` block need be CFML functions. In the previous example, literal text was used, too, and that text was sent to the client untouched. As such, you also could have entered the code like this:

```
It is now <CFOUTPUT>#Now()#</CFOUTPUT>
```

Only the `#Now()#` expression needs ColdFusion processing, so only it really needs to be within the `<CFOUTPUT>` block. But what if you had not placed the expression within a `<CFOUTPUT>` block? Try it—remove the `<CFOUTPUT>` tags, save the page, and execute it. You'll see output similar to that in Figure 9.4—obviously not what you want. Because any content not within a `<CFOUTPUT>` block is sent to the client as is, using `Now()` outside a `<CFOUTPUT>` block causes the text `Now()` to be sent to the client instead of the data returned by `Now()`. Why? Because if it's outside a `<CFOUTPUT>` block, ColdFusion will never process it.

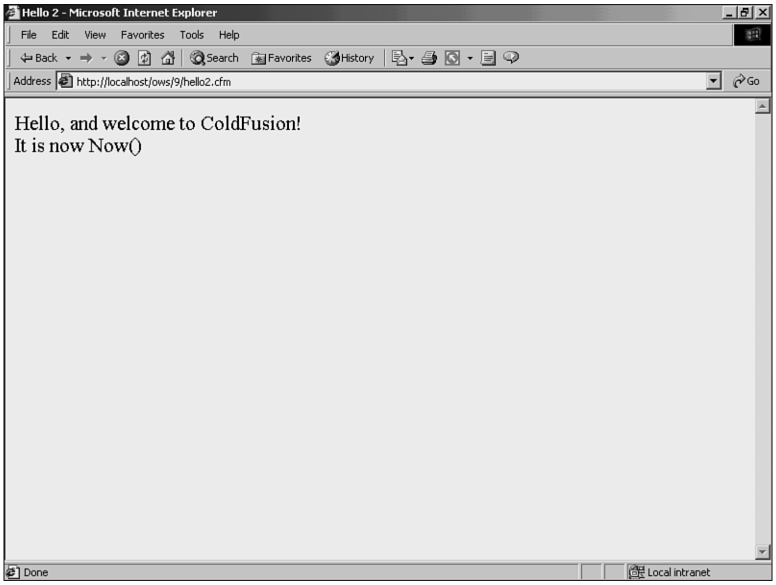

Hello 2 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello2.cfm
Go
Done
Local intranet

```
Hello, and welcome to ColdFusion!
It is now #Now()#
```

Figure 9.4
If expressions are sent to the browser, it usually means you have omitted the `<CFOUTPUT>` tags.

Omitting the pound signs has a similar effect. Put the `<CFOUTPUT>` tags back where they belong, but change `#Now()#` to `Now()` (removing the pound signs from before and after it). Then save the page, and execute it. The output will look similar to Figure 9.5. Why? Because all `<CFOUTPUT>` does is flag a block of text as needing processing by ColdFusion.

However, ColdFusion does not process *all* text between the tags—instead, it looks for expressions delimited by pound signs, and any text *not* within pound signs is assumed to be literal text that is to be sent to the client as is.


Hello 2 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello2.cfm
Go
Done
Local intranet

Hello, and welcome to ColdFusion!
It is now Now()

Figure 9.5
# signs are needed around all expressions; otherwise, the expression is sent to the client instead of being processed.

Note

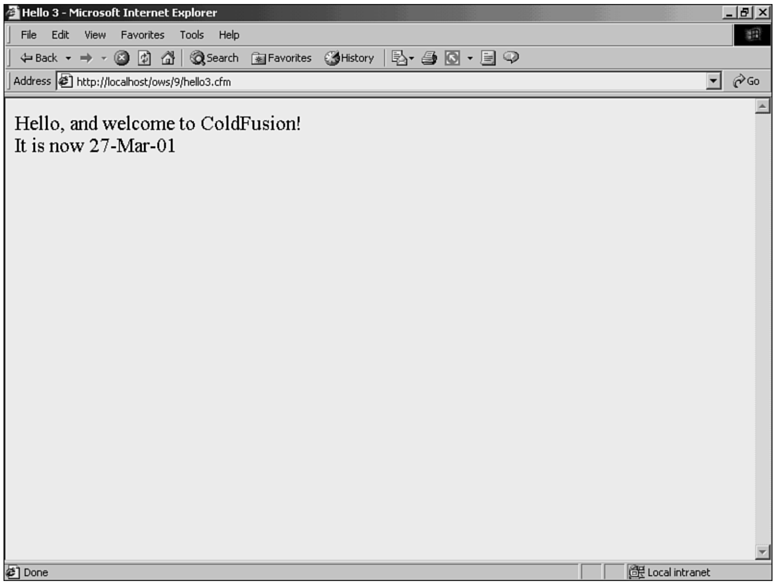
`<CFOUTPUT>` has another important use when working with database-driven content. More information about that can be found in Chapter 11, "Creating Data-Driven Pages."

`Now()` is a function, one of many functions supported in CFML. `Now()` is used to retrieve information from the system (the date and time), but the format of that date is not entirely readable. Another function, `DateFormat()`, can help here. `DateFormat()` is one of ColdFusion's output formatting functions, and its job is to format dates so they are readable (in all types of formats). Listing 9.3 is a revision of the code you just used; save it as `hello3.cfm` and browse the file to see output similar to what is shown in Figure 9.6.

Listing 9.3 `hello3.cfm`

```
<HTML>
<HEAD>
   <TITLE>Hello 3</TITLE>
</HEAD>

<BODY>
```

```
Hello, and welcome to ColdFusion!
<BR>
<CFOUTPUT>
It is now #DateFormat(Now())#
</CFOUTPUT>

</BODY>
</HTML>
```


Hello 3 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello3.cfm
Go
Done
Local intranet

Hello, and welcome to ColdFusion!
It is now 27-Mar-01

Figure 9.6 ColdFusion features a selection of output formatting functions that can be used to better control generated output.

`DateFormat()` is an example of a function that accepts (and requires) that data must be passed to it—after all, it needs to know which date you want to format for display. `DateFormat()` can accept dates as hard-coded strings (as in `#DateFormat("12/13/01")#`), as well as dates returned by other expressions, such as the `Now()` function. `#DateFormat(Now())#` tells ColdFusion to format the date returned by the `Now()` function.

Note

Passing a function as a parameter to another function is referred to as *nesting*. In this chapter's example, the `Now()` function is said to be *nested* in the `DateFormat()` function.

`DateFormat()` takes a second optional attribute, too: a format mask used to describe the output format. Try replacing the `#DateFormat(Now())#` in your code with any of the following (and try each to see what they do):

- `#DateFormat(Now(), "MMMM-DD-YYYY")#`
- `#DateFormat(Now(), "MM/DD/YY")#`
- `#DateFormat(Now(), "DDD, MMMM DD, YYYY")#`

Parameters passed to a function are always separated by commas. Commas are not used if a single parameter is passed, but when two or more parameters exist, every parameter must be separated by a comma.

You've now seen a function that takes no parameters, a function that takes a required parameter, and a function that takes both required and optional parameters. All ColdFusion functions, and you'll be using many of them, work the same way—some take parameters, and some do not. But all functions, regardless of parameters, return a value.

Note

It is important to remember that `#` is not part of the function—the functions you used here were `DateFormat()` and `Now()`. The pound signs were used to delimit (mark) the expressions, but they are not part of the expression itself.

I know I have already said this, but it is worth repeating—CFML code is processed on the server, not on the client. The CFML code you write is *never* sent to the Web browser. What is sent to the browser? Most browsers feature a View Source option that displays code as received. If you view the source of for page `hello3.cfm` you'll see something like this:

```
<HTML>
<HEAD>
   <TITLE>Hello 3</TITLE>
</HEAD>

<BODY>

Hello, and welcome to ColdFusion!
<BR>

It is now 27-Mar-01


</BODY>
</HTML>
```

As you can see, there is no CFML code here at all. The `<CFOUTPUT>` tags, the functions, the pound signs—all have been stripped out by the ColdFusion Server, and what was sent to the client is the output that they generated.

Tip

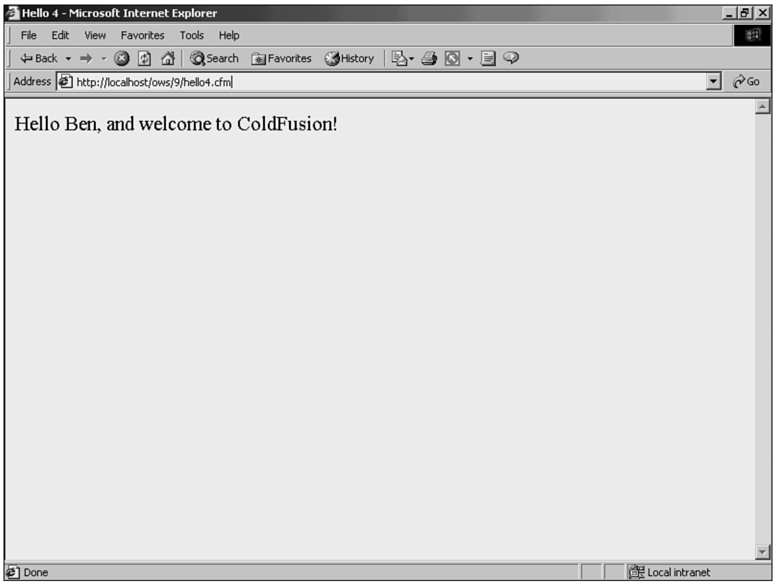
Viewing the generated source is an invaluable debugging trick. If you ever find that output is not being generated as expected, viewing the source can help you understand exactly what was generated and why.

USING VARIABLES

Now that you've had the chance to use some basic functions, it's time to introduce variables. Variables are an important part of just about every programming language, and CFML is no exception. A *variable* is a container that stores information in memory on the server. Variables are named, and the contents of the container are accessed via that name. Let's look at a simple example, seen in Listing 9.4. Type the code, save it as `hello4.cfm`, and browse it. You should see a display similar to the one shown in Figure 9.7.

LISTING 9.4 `hello4.cfm`

```
<HTML>
<HEAD>
   <TITLE>Hello 4</TITLE>
</HEAD>

<BODY>

<CFSET FirstName="Ben">
<CFOUTPUT>
Hello #FirstName#, and welcome to ColdFusion!
</CFOUTPUT>

</BODY>
</HTML>
```


Hello 4 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello4.cfm
Go
Done
Local intranet

Hello Ben, and welcome to ColdFusion!

Figure 9.7
Variables are replaced by their contents when content is generated.

The code in Listing 9.4 is similar to the previous code listings. It starts with plain HTML that is sent to the client as is. Then a new tag is used, `<CFSET>`:

```
<CFSET FirstName="Ben">
```

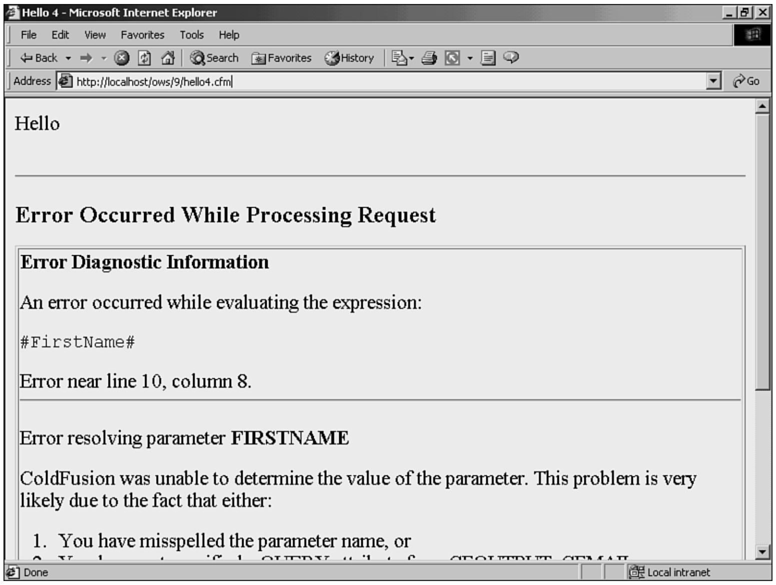
`<CFSET>` is used to set variables. Here, a variable named `FirstName` is created, and a value of `"Ben"` is stored in it. After it's created, that variable will exist until the page has finished processing and can be used as seen in the next line of code:

```
Hello #FirstName#, and welcome to ColdFusion!
```

This line of code was placed in a `<CFOUTPUT>` block so ColdFusion would know to replace `#FirstName#` with the contents of `FirstName`. The generated output is then

```
Hello Ben, and welcome to ColdFusion!
```

Variables can be used as many times as necessary, as long as they exist. Try moving the `<CFSET>` statement after the `<CFOUTPUT>` block, or delete it altogether. Executing the page now will generate an error, similar to the one seen in Figure 9.8. This error message is telling you that you referred to (tried to access) a variable that does not exist. The error message tells you the name of the variable that caused the problem, as well as the line and column in your code (to help you find and fix the problem easily). More often than not, this error is caused by typos.


Hello 4 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello4.cfm
Go
Done
Local intranet

Hello

Error Occurred While Processing Request

Error Diagnostic Information

An error occurred while evaluating the expression:

```
#FirstName#
```

Error near line 10, column 8.

Error resolving parameter **FIRSTNAME**

ColdFusion was unable to determine the value of the parameter. This problem is very likely due to the fact that either:

1. You have misspelled the parameter name, or

Figure 9.8
ColdFusion produces an error if a referenced variable does not exist.

Note

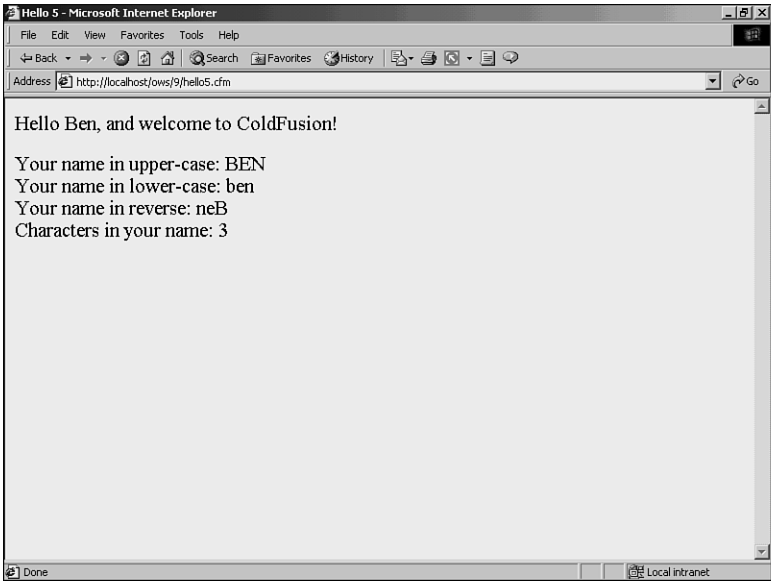
Regular variables exist only in the page that creates them. If you define a variable named `FirstName` in one page, you can't use it in another page unless you explicitly pass it to that page (see Chapter 11).

An exception to this rule does exist. In Chapter 20, "Working with Sessions," you learn how to create and use variables that persist across requests (each page access is known as a *request*).

Listing 9.5 contains a new version of the code, this time using the variable `FirstName` five times. Try this listing for yourself (feel free to replace my name with your own). The output is shown in Figure 9.9.

LISTING 9.5 `hello5.cfm`

```
<HTML>
<HEAD>
   <TITLE>Hello 5</TITLE>
</HEAD>

<BODY>

<CFSET FirstName="Ben">
<CFOUTPUT>
Hello #FirstName#, and welcome to ColdFusion!<P>
Your name in uppercase: #UCase(FirstName)#<BR>
Your name in lowercase: #LCase(FirstName)#<BR>
Your name in reverse: #Reverse(FirstName)#<BR>
Characters in your name: #Len(FirstName)#<BR>
</CFOUTPUT>

</BODY>
</HTML>
```


Hello 5 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello5.cfm
Go
Done
Local intranet

```
Hello Ben, and welcome to ColdFusion!

Your name in upper-case: BEN
Your name in lower-case: ben
Your name in reverse: neB
Characters in your name: 3
```

Figure 9.9
There is no limit to the number of functions that can be used in one page, which enables you to render content as you see fit.

Let's take a look at the code in Listing 9.5. A `<CFSET>` is used to create a variable named `FirstName`. That variable is used once by itself (the Hello message), and then four times with functions. `UCase()` converts a string to uppercase, `LCase()` converts a string to lowercase, `Reverse()` reverses the string, and `Len()` returns the length of a string (the number of characters in it).

But functions such as `UCase()` don't truly convert strings; instead, they return converted strings. The difference is subtle but important. Look at the following line of code:

```
Your name in uppercase: #UCase(FirstName)#
```

`UCase()` returns `FirstName` converted to uppercase, but the contents of `FirstName` itself are intact. `FirstName` was not modified; a copy was made and modified instead, and that copy was returned. To save the uppercase `FirstName` to a variable, you must do something like this:

```
<CFSET UpperFirstName=UCase(FirstName)>
```

Here a new variable, `UpperFirstName`, is created. `UpperFirstName` is assigned the value that is returned by `UCase(FirstName)`, the uppercase `FirstName`. And this new variable can be used like any other variable, and as often as necessary. Listing 9.6 contains a modified version of the previous listing. Try it for yourself—the output will be exactly the same as in Figure 9.9.

LISTING 9.6 `hello6.cfm`

```
<HTML>
<HEAD>
   <TITLE>Hello 6</TITLE>
</HEAD>

<BODY>

<CFSET FirstName="Ben">
<CFSET UpperFirstName=UCase(FirstName)>
<CFSET LowerFirstName=LCase(FirstName)>
<CFSET ReverseFirstName=Reverse(FirstName)>
<CFSET LenFirstName=Len(FirstName)>

<CFOUTPUT>
Hello #FirstName#, and welcome to ColdFusion!<P>
Your name in uppercase: #UpperFirstName#<BR>
Your name in lowercase: #LowerFirstName#<BR>
Your name in reverse: #ReverseFirstName#<BR>
Characters in your name: #LenFirstName#<BR>
</CFOUTPUT>

</BODY>
</HTML>
```

This code deserves a look. Five `<CFSET>` tags now exist, and five variables are created. The first creates the `FirstName` variable, just like in the previous examples. The next creates a new variable named `UpperFirstName`, which contains the uppercase version of `FirstName`. And then `LowerFirstName`, `ReverseFirstName`, and `LenFirstName` are each created with additional `<CFSET>` statements.

The `<CFOUTPUT>` block here contains no functions at all. Rather, it just displays the contents of the variables that were just created. In this particular listing there is actually little value in doing this, aside from the fact that the code is a little more organized this way. The real benefit in saving function output to variables is realized when a function is used many times in a single page. Then, instead of using the same function over and over, you can use it once, save the output to a variable, and just use that variable instead.

One important point to note here is that variables can be overwritten. Look at the following code snippet:

```
<CFSET FirstName="Ben">
<CFSET FirstName="Nate">
```

Here, `FirstName` is set to `Ben` and then set again to `Nate`. Variables can be overwritten as often as necessary, and whatever the current value is when accessed (displayed, or passed to other functions), that's the value that will be used.

Knowing that, what do you think the following line of code does?

```
<CFSET FirstName=UCase(FirstName)>
```

This is an example of variable overwriting, but here the variable being overwritten is the variable itself. I mentioned earlier that functions such as `UCase()` do not convert text; they return a converted copy. So how could you really convert text? By using code such as the line just shown. `<CFSET FirstName=UCase(FirstName)>` sets `FirstName` to the uppercase version of `FirstName`, effectively overwriting itself with the converted value.

VARIABLE NAMING

This would be a good place to discuss variable naming. When you create a variable you get to name it, and the choice of names is up to you. However, you need to know a few rules about variable naming:

- Variable names can contain alphanumeric characters but can't begin with a number (so `result12` is okay, but `4thresult` is not).
- Variable names can't contain spaces. If you need to separate words, use underscores (for example, `monthly_sales_figures` instead of `monthly sales figures`).
- Aside from the underscore, nonalphanumeric characters can't be used in variable names (so `Sales!`, `SSN#`, and `first-name` are all invalid).
- Variable names are case insensitive (`FirstName` is the same as `FIRSTNAME` which is the same as `firstname`).

Other than that, you can be as creative as necessary with your names. Pick any variable name you want; just be careful not to overwrite existing variables by mistake.

Tip

Avoid the use of abbreviated variable names, such as `fn` or `c`. Although these are valid names, what they stand for is not apparent just by looking at them. Yes, `fn` is less

keystrokes than `FirstName`, but the first time you (or someone else) has to stare at the code trying to figure out what a variable is for, you'll regret saving that little bit of time. As a rule, make variable names descriptive.

Using Prefixes

ColdFusion supports many variable types, and you'll become very familiar with them as you work through this book. For example, local variables (the type you just created) are a variable type. Submitted form fields are a variable type, as are many others.

ColdFusion variables can be referenced in two ways:

- The variable name itself
- The variable name with the type as a prefix

For example, the variable `FirstName` that you used a little earlier is a local variable (type `VARIABLES`). So, that variable can be referred to as `FirstName` (as you did previously) and as `VARIABLES.FirstName`. Both are valid, and both will work (you can try editing file `hello6.cfm` to use the `VARIABLES` prefix to try this).

So, should you use prefixes? Well, there are pros and cons. Here are the pros:

- Using prefixes improves performance. ColdFusion will have less work to do finding the variable you are referring to if you explicitly provide the full name (including the prefix).
- If multiple variables exist with the same name but are of different types, the only way to be 100% sure that you'll get the variable you want is to use the prefix.

As for the cons, there is just one:

- If you omit the prefix then multiple variable types will be accessible (perhaps form fields and URL parameters, which are discussed in the following chapters). If you provide the type prefix, you restrict access to the specified type, and although this does prevent ambiguity (as just explained), it does make your code a little less reusable.

The choice is yours, and there is no real right or wrong. You can use prefixes if you see fit, and not use them if not. If you don't specify the prefix, ColdFusion will find the variable for you. And if multiple variables of the same name do exist (with differing types) then a predefined order of precedence is used (don't worry if these types are not familiar yet, they will become familiar soon enough, and you can refer to this list when necessary):

- Query results
- Local variables (`VARIABLES`)
- `CGI` variables
- `FILE` variables

- `URL` parameters
- `FORM` fields
- `COOKIE` values
- `CLIENT` variables

In other words, if you refer to `#FirstName#` (without specifying a prefix) and that variable exists both as a local variable (`VARIABLES.FirstName`) and as a `FORM` field (`FORM.FirstName`), `VARIABLES.FirstName` will be used automatically.

Note

An exception to this does exist. Some ColdFusion variable types must *always* be accessed with an explicit prefix; these are covered in later chapters.

WORKING WITH EXPRESSIONS

I've used the term *expressions* a few times in this chapter. What is an expression? The official ColdFusion documentation explains that expressions are "language constructs that allow you to create sophisticated applications." A better way to understand it is that expressions are strings of text made up of one or more of the following:

- Literal text (strings), numbers, dates, times, and other values
- Variables
- Operators (`+` for addition, `&` for concatenation, and so on)
- Functions

So, `UCase(FirstName)` is an expression, as is `"Hello, my name is Ben"`, `12+4`, and `DateFormat(Now())`. And even though many people find it hard to articulate exactly what an expression is, realize that expressions are an important part of the ColdFusion language.

BUILDING EXPRESSIONS

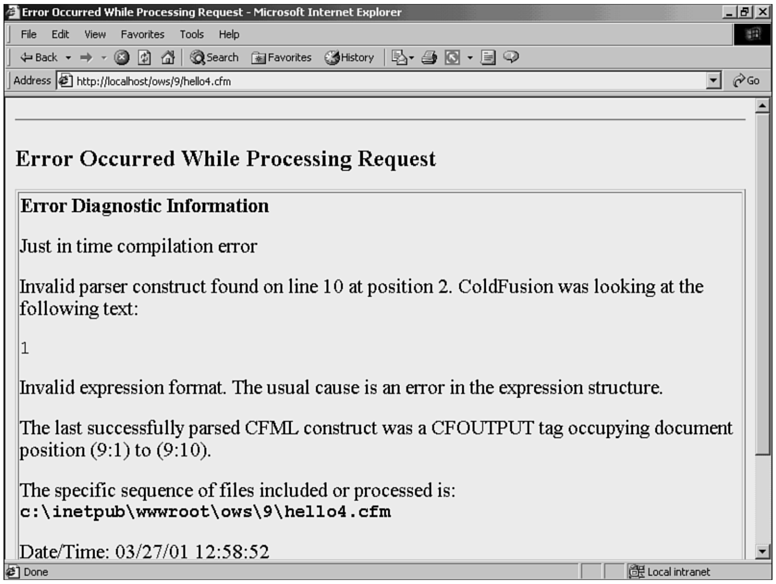
Expressions are typed where necessary. Expressions can be passed to a `<CFSET>` statement as part of an assignment, used when displaying text, and passed to almost every single CFML tag (except for the few that take no attributes).

Simple expressions can be used, such as those discussed previously (variables, functions, and combinations thereof). But more complex expressions can be used, too, and expressions can include arithmetic, string, and decision operators (you'll use these in the next few chapters).

When using expressions, pound signs are used to delimit ColdFusion functions and variables within a block of text. So, how would you display the `#` itself? Look at the following code snippet:

```
<CFOUTPUT>
#1: #FirstName#
</CFOUTOUT>
```

You can try this yourself if you so feel inclined; you'll see that ColdFusion generates an error when it processes the code (see Figure 9.10).


Error Occurred While Processing Request - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello4.cfm
Go
Done
Local intranet

Error Occurred While Processing Request

Error Diagnostic Information

Just in time compilation error

Invalid parser construct found on line 10 at position 2. ColdFusion was looking at the following text:

```
1
```

Invalid expression format. The usual cause is an error in the expression structure.

The last successfully parsed CFML construct was a CFOUTPUT tag occupying document position (9:1) to (9:10).

The specific sequence of files included or processed is:
c:\inetpub\wwwroot\ows\9\hello4.cfm

Date/Time: 03/27/01 12:58:52

Figure 9.10
Pound signs in text must be escaped; otherwise, ColdFusion produces an error.

What causes this error? Well, when ColdFusion encounters the # at the start of the line, it assumes you are delimiting a variable or a function and tries to find the matching # (which of course does not exist). The solution is to *escape* the pound sign (flag it as being a real pound sign), as follows:

```
<CFOUTPUT>
##1: #FirstName#
</CFOUTOUT>
```

When ColdFusion encounters ##, it knows that # is not delimiting a variable or function. Instead, it correctly displays a single #.

WHEN TO USE #, AND WHEN NOT TO

Before we go any further, let's clarify exactly when pound signs are needed and when they're not.

Simply put, pound signs are needed to flag functions and variables within a string of text.

So, in this first example, the pound signs are obviously needed:

```
Hello #VARIABLES.FirstName#
```

But what about when a variable is used within a tag, like this?

```
<CFSET UpperFirstName=UCase(FirstName)>
```

Here pound signs are not necessary because ColdFusion assumes that anything passed to a tag is a function or variable unless explicitly defined as a string. So, the following is incorrect:

```
<CFSET #UpperFirstName#=#UCase(FirstName)#>
```

This code will actually work (ColdFusion is very forgiving), but it is still incorrect and should not be used.

This next example declares a variable and assigns a value that is a string, so no pound signs are needed here:

```
<CFSET FirstName="Ben">
```

But if the string contains variables, pound signs would be necessary. Look at this next example: `FullName` is assigned a string, but the string contains two variables (`FirstName` and `LastName`) and those variables must be enclosed within pound signs (otherwise ColdFusion will assign the text, not the variable values):

```
<CFSET FullName="#FirstName# #LastName#">
```

Incidentally, the previous line of code is functionally equivalent to the following:

```
<CFSET FullName=FirstName & " " & LastName>
```

Here pound signs are not necessary because the variables are not being referred to within a string.

Again, the rule is this—only use pound signs when referring to variables and functions within a block of text. Simple as that.

USING COLDFUSION DATA TYPES

The variables you have used thus far are simple variables, are defined, and contain a value. ColdFusion supports three advanced data types that I'll briefly introduce now.

Note

This is just an introduction to lists, arrays, and structures. All three are used repeatedly throughout the rest of this book, so don't worry if you do not fully understand them by the time you are done reading this chapter. Right now, the intent is to ensure that you know these exist and what they are. You'll have lots of opportunities to use them soon enough.

LISTS

Lists are used to group together related information. Lists are actually strings (plain text)—what makes them lists is that a delimiter is used to separate items within the string. For example, the following is a comma-delimited list of five U.S. states:

```
California,Florida,Michigan,Massachusetts,New York
```

The next example is also a list. Even though it might not look like a list, a sentence is a list delimited by spaces:

```
This is a ColdFusion list
```

Lists are created just like any other variables. For example, this next line of code uses the `<CFSET>` tag to create a variable named `fruit` that contains a list of six fruits:

```
<CFSET fruit="apple,banana,cherry,grape,mango,orange">
```

Listing 9.7 demonstrates the use of lists. Type the code and save it as `list.cfm` in the `9` directory; then execute it. You should see an output similar to the one shown in Figure 9.11.

Listing 9.7 `list.cfm`

```
<HTML>
<HEAD>
   <TITLE>List Example</TITLE>
</HEAD>

<BODY>

<CFSET fruit="apple,banana,cherry,grape,mango,orange">
<CFOUTPUT>
Complete list: #fruit#<BR>
Number of fruit in list: #ListLen(fruit)#<BR>
First fruit: #ListFirst(fruit)#<BR>
Last fruit: #ListLast(fruit)#<BR>
<CFSET fruit=ListAppend(fruit, "pineapple")>
Complete list: #fruit#<BR>
Number of fruit in list: #ListLen(fruit)#<BR>
First fruit: #ListFirst(fruit)#<BR>
Last fruit: #ListLast(fruit)#<BR>
</CFOUTPUT>

</BODY>
</HTML>
```

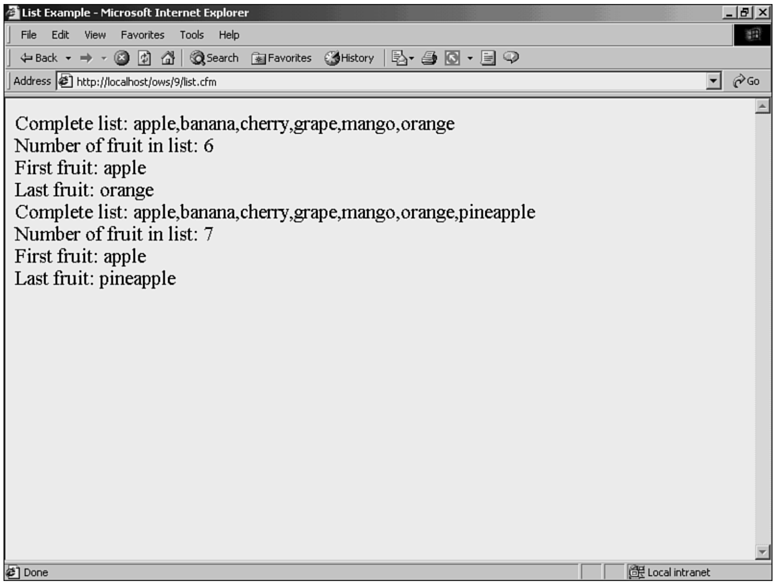
Let's walk through the code. A `<CFSET>` is used to create a list, which is simply a string. Therefore, a simple variable assignment can be used.

Next comes the `<CFOUTPUT>` block, starting with displaying `#fruit#` (the complete list). The next line of code uses the `ListLen()` function to return the number of items in the list (there are `6` of them). Individual list members can be retrieved using `ListFirst()` (used here to get the first list element), `ListLast()` (used here to get the last list element), and `ListGetAt()` (used to retrieve any list element, but not used in this example).

Then another `<CFSET>` tag is used, as follows:

```
<CFSET fruit=ListAppend(fruit, "pineapple")>
```

This code uses the `ListAppend()` function to add an element to the list. You will recall that functions return copies of modified variables, not modified variables themselves. So the `<CFSET>` tag assigns the value returned by `ListAppend()` to `fruit`, effectively overwriting the list with the new revised list.



```
Complete list: apple,banana,cherry,grape,mango,orange
Number of fruit in list: 6
First fruit: apple
Last fruit: orange
Complete list: apple,banana,cherry,grape,mango,orange,pineapple
Number of fruit in list: 7
First fruit: apple
Last fruit: pineapple
```

Figure 9.11 Lists are useful for grouping related data into simple sets.

Then the number of items, as well as the first and last items, are displayed again. This time `7` items are in the list, and the last item has changed to `pineapple`.

As you can see, lists are very easy to use and provide a simple mechanism for grouping related data.

Note

I mentioned earlier that a sentence is a list delimited by spaces. The default list delimiter is indeed a comma. Actually, though, *any* character can be used as a list delimiter, and every list function takes an optional delimiter attribute if necessary.

ARRAYS

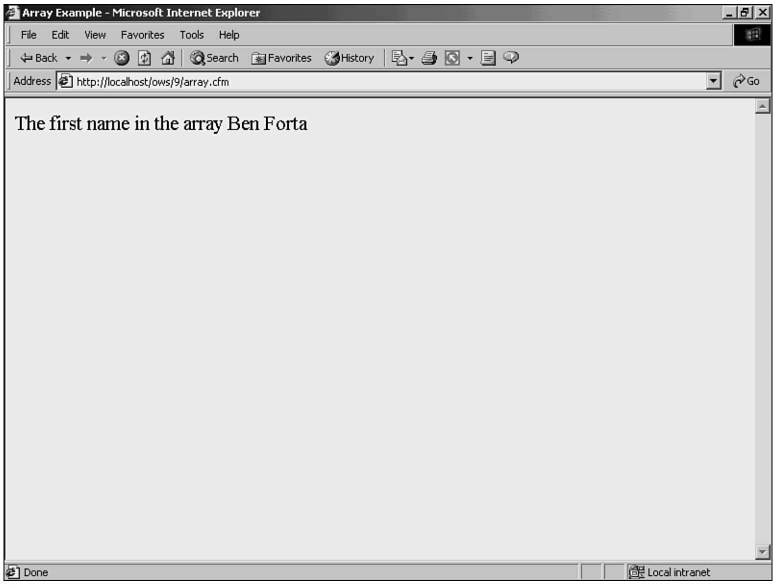
Arrays, like lists, store multiple values in a single variable. But unlike lists, arrays can contain far more complex data (including lists and even other arrays).

Unlike lists, arrays support multiple dimensions. A single dimensional array is actually quite similar to a list: It's a linear collection. A two-dimensional array is more like a grid (imagine a spreadsheet), and data is stored in rows and columns. ColdFusion also supports three-dimensional arrays, which can be envisioned as cubes of data.

If this all sounds somewhat complex, well, it is. Arrays are not as easy to use as lists are, but they are far more powerful (and far quicker). Listing 9.8 contains a simple block of code that creates an array and displays part of it; the output is shown in Figure 9.12. To try it out, type the code and save it as `array.cfm`.

LISTING 9.8 `array.cfm`

```
<HTML>
<HEAD>
   <TITLE>Array Example</TITLE>
</HEAD>

<BODY>

<CFSET names=ArrayNew(2)>
<CFSET names[1][1]="Ben">
<CFSET names[1][2]="Forta">
<CFSET names[2][1]="Nate">
<CFSET names[2][2]="Weiss">

<CFOUTPUT>
The first name in the array #names[1][1]# #names[1][2]#
</CFOUTPUT>

</BODY>
</HTML>
```


Array Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/array.cfm
Go
Done
Local intranet

The first name in the array Ben Forta

Figure 9.12
Arrays treat data as if it were in a one-, two-, or three-dimensional grid.

Arrays are created using the `ArrayNew()` function. `ArrayNew()` requires that the desired dimension be passed as a parameter, so the following code creates a two-dimensional array named `names`:

```
<CFSET names=ArrayNew(2)>
```

Array elements are set using `<CFSET>`, just like any other variables. But unlike other variables, when array elements are set the element number must be specified using an index (a relative position starting at 1). So, in a single dimensional array, names[1] would refer to the first element and names[6] would refer to the sixth. In two-dimensional arrays, both dimensions must be specified, as seen in these next four lines (taken from Listing 9.8):

```
<CFSET names[1][1]="Ben">
<CFSET names[1][2]="Forta">
<CFSET names[2][1]="Nate">
<CFSET names[2][2]="Weiss">
```

names[1][1] refers to the first element in the first dimension—think of it as the first column of the first row in a grid. names[1][2] refers to the second column in that first row, and so on.

When accessed, even for display, the indexes must be used. Therefore, the following line of code

```
The first name in the array #names[1][1]# #names[1][2]#
```

generates this output:

```
The first name in the array Ben Forta
```

As you can see, although they're not as easy to use as lists, arrays are a very flexible and powerful language feature.

STRUCTURES

Structures are the most powerful and flexible data type within ColdFusion, so powerful in fact that many internal variables (including ones listed in Appendix C, "Special ColdFusion Variables and Result Codes") are structures internally.

Simply put, structures provide a way to store data within data. Unlike arrays, structures have no special dimensions and are not like grids. Rather, they can be thought of as top-level folders that can store data, or other folders, which in turn can store data, or other folders, and so on. Structures can contain lists, arrays, and even other structures.

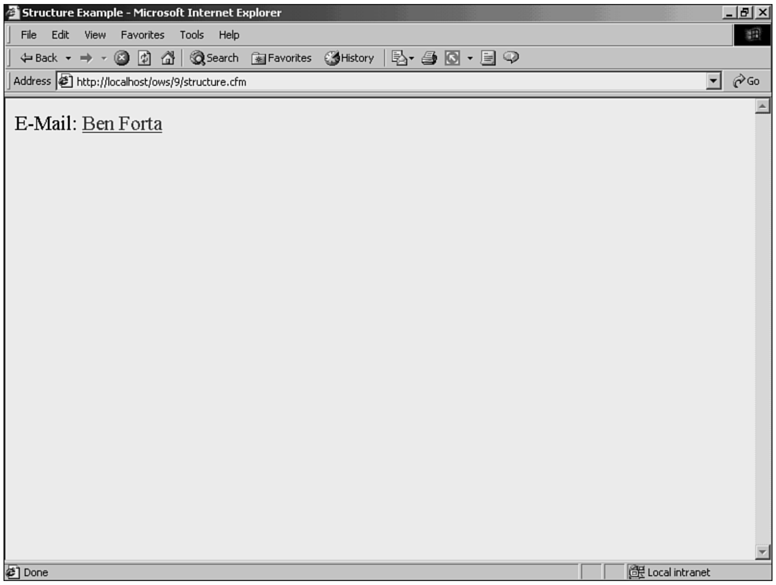
To give you a sneak peek at what structures look like, see Listing 9.9. Give it a try yourself; save the file as structure.cfm, and you should see output as shown in Figure 9.13.

LISTING 9.9 structure.cfm

```
<HTML>
<HEAD>
   <TITLE>Structure Example</TITLE>
</HEAD>

<BODY>

<CFSET contact=StructNew()>
<CFSET contact.FirstName="Ben">
<CFSET contact.LastName="Forta">
<CFSET contact.EMail="ben@forta.com">
```

LISTING 9.9 CONTINUED

```
<CFOUTPUT>
E-Mail:
<A HREF="mailto:#contact.EMail#">#contact.FirstName# #contact.LastName#</A>
</CFOUTPUT>

</BODY>
</HTML>
```


Structure Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/structure.cfm
Go
Done
Local intranet

E-Mail: Ben Forta

Figure 9.13
Structures are the most powerful data type in ColdFusion and are used internally extensively.

Structures are created using `StructNew()`, which—unlike `ArrayNew()`—takes no parameters. After a structure is created, variables can be set inside it. The following three lines of code all set variables with the `contact` structure:

```
<CFSET contact.FirstName="Ben">
<CFSET contact.LastName="Forta">
<CFSET contact.EMail="ben@forta.com">
```

To access structure members, simply refer to them by name. `#contact.FirstName#` accesses the `FirstName` member of the `contact` structure. Therefore, the code

```
<A HREF="mailto:#contact.EMail#">#contact.FirstName# #contact.LastName#</A>
```

generates this output:

```
<A HREF="mailto:ben@forta.com">Ben Forta</A>
```

And that's just scratching the surface. Structures are incredibly powerful, and you'll use them extensively as you work through this book.

Note

For simplicity's sake, I have described only the absolute basic form of structure use. ColdFusion features an entire set of structure manipulation functions that can be used to better take advantage of structures—you use some of them in the next chapter, "CFML Basics."

Commenting Your Code

The last introductory topic I want to mention is commenting your code. Many books leave this to the very end, but I believe it is so important that I am introducing the concept right here—before you start real coding.

The code you have worked with thus far has been short, simple, and rather self-explanatory. But as you start building bigger and more complex applications, your code will become more involved and more complex. And then comments become vital. The reasons to comment your code include

- If you make code as self-descriptive as possible, when you revisit it at a later date you'll remember what you did, and why.
- This is even truer if others have to work on your code. The more detailed and accurate comments are, the easier (and safer) it will be to make changes or corrections when necessary.
- Commented code is much easier to debug than uncommented code.
- Commented code tends to be better organized, too.

And that's just the start of it.

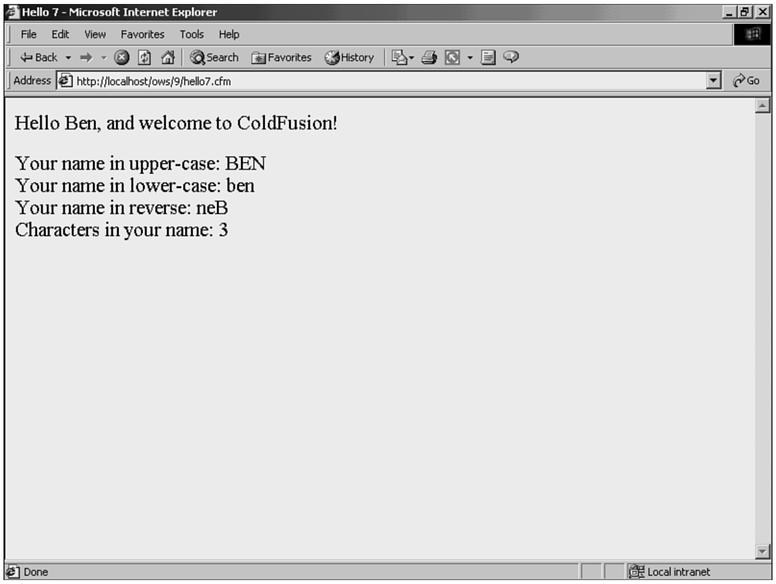
Listing 9.10 is a revised version of `hello6.cfm`; all that has changed is the inclusion of comments. And as you can see from Figure 9.14, this has no impact on generated output whatsoever.

Listing 9.10 `hello7.cfm`

```
<!---
Name:        hello7.cfm
Author:      Ben Forta (ben@forta.com)
Description: Demonstrate use of comments
Created:     3/27/01
--->

<HTML>
<HEAD>
   <TITLE>Hello 7</TITLE>
</HEAD>

<BODY>
```

Listing 9.10 Continued

```
<!--- Save name --->
<CFSET FirstName="Ben">

<!--- Save converted versions of name --->
<CFSET UpperFirstName=UCase(FirstName)>
<CFSET LowerFirstName=LCase(FirstName)>
<CFSET ReverseFirstName=Reverse(FirstName)>

<!--- Save name length --->
<CFSET LenFirstName=Len(FirstName)>

<!--- Display output --->
<CFOUTPUT>
Hello #FirstName#, and welcome to ColdFusion!<P>
Your name in uppercase: #UpperFirstName#<BR>
Your name in lowercase: #LowerFirstName#<BR>
Your name in reverse: #ReverseFirstName#<BR>
Characters in your name: #LenFirstName#<BR>
</CFOUTPUT>

</BODY>
</HTML>
```


Hello 7 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost/ows/9/hello7.cfm
Go
Done
Local intranet

```
Hello Ben, and welcome to ColdFusion!

Your name in upper-case: BEN
Your name in lower-case: ben
Your name in reverse: neB
Characters in your name: 3
```

Figure 9.14
ColdFusion comments in your code are never sent to the client browser.

Comments are typed between `<!---` and `--->` tags. Comments should never be nested and should never be mismatched (such as having a starting tag without an end tag, or vice versa).

Note

ColdFusion uses `<!---` and `--->` to delimit comments. HTML uses `<!--` and `-->` (two hyphens instead of three). Within ColdFusion code, always use ColdFusion comments and not HTML comments. The latter will be sent to the client (they won't be displayed, but they will still be sent), whereas the former won't.

Tip

Commenting code is a useful debugging technique. When you are testing code and need to eliminate specific lines, you can *comment them out* temporarily by wrapping them within `<!---` and `--->` tags.