Creating field help for your Domino applications (part 2)

by Mark Gordon and Micah Blalock

*[Editor's note: This article resides in "Iris Today", the technical Webzine located on the http://www.notes.net Web site produced by Iris Associates, the developers of Domino and Notes. This is the second article in a two-part series on how to create context-sensitive field help for your Domino applications. This article focuses on how to create reusable help topics, and looks at an alternative field help solution for Netscape 4.x browsers. The **first article** showed you how to provide field help in a separate help frame, basic field help that appears when users tab into a field, and pop-up help.]*

When designing your Web applications, field help may seem like one of those non-essential "extra's." However, it's an "extra" that your users will really appreciate! You may not realize that you can easily create field help that does more than display in the status line of a browser. This article shows you some techniques for making field help show up in any font and size you like so that users can't miss it.

In the **first article** in this two-part series, we introduced you to three techniques for adding field help to your applications. First, we showed you how to easily incorporate a small help frame at the bottom of your forms, without needing to redesign your application to account for frames. Our second technique demonstrated how to create basic field help that displays when users tab into a field, or move their mouse over a field label. Finally, we showed you how to add pop-up help windows to your application for displaying longer help descriptions.

In this second article, you'll learn about two additional field help techniques. We'll first look at how to create field help that you can reuse for multiple fields in your application. The technique works by storing the help text separately. You'll also learn how to use Domino to generate portions of the JavaScript for you. Then, the final technique we'll cover is an alternative field help solution for Netscape 4.x browsers, which have a tabbing problem when JavaScript is used in field events. Our work-around shows you how to generate different JavaScript calls for different browsers. All the examples we'll discuss are working examples you can try by downloading our sample database.

Downloading the sample database

You can try out the techniques described in this article by downloading the following self-extracting database (211Kb):

fieldhp2.exe

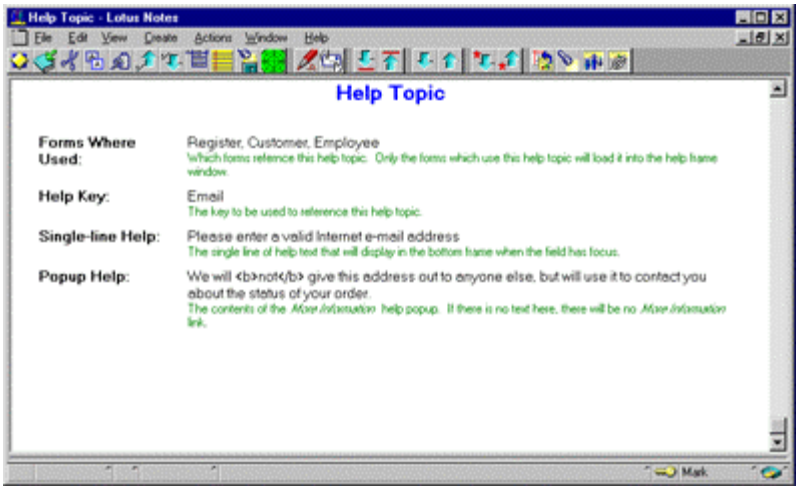
You can refer to this database as you read through the rest of this article, and then use it to create field help in your own applications. For information on how to use this database to create your own field help, see the sidebar "**Quick steps to adding field help to your applications**."

Reusing field help

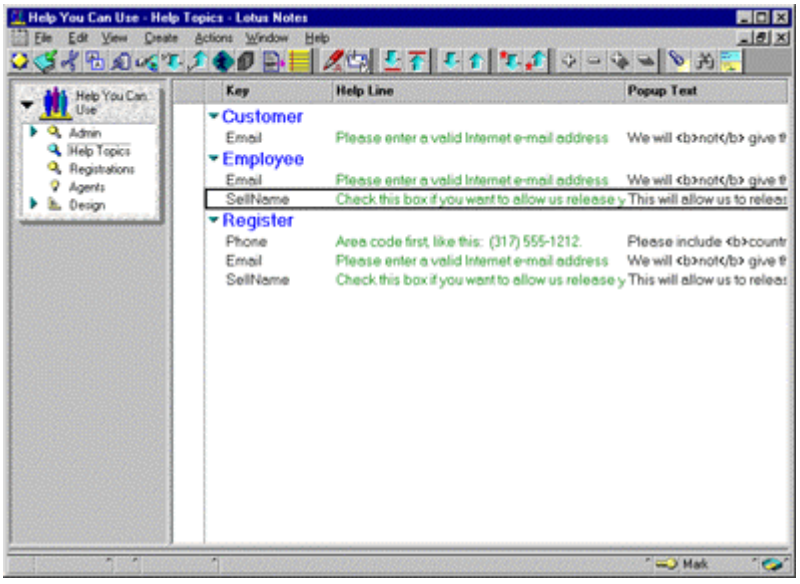
When you begin creating field help for your application, you may find that you have similar fields that could display the same help text. For example, you might want to display the same tip on formatting a phone number -- such as "area code first please" -- with multiple phone number fields, fax number fields, and so on. In the **first article**, we used the FrameHelpText function to directly display any help text. This time, we'll use the FrameHelpKey function (also defined on the Register form), which is very much like the FrameHelpText function, except that it uses reusable field help to display help text in the help frame. It accepts a help "key" argument, rather than directly passing the help text into the function. For example, here is what the HTML Attributes formula looks like for the Phone field, which uses a reusable help topic called *Phone:*

```
HelpKey := "Phone";
"onFocus=\'FrameHelpKey(\"" + HelpKey + "\")\'"
```

Before we discuss how the FrameHelpKey function works, let's take a look at how we create and maintain the reusable help text. We first define a help topic form, which the developers and/or application owners can use to define and maintain reusable help topic documents. For example, here's a help topic document for the e-mail field:



Notice that we've defined this topic to be used on three forms: Register, Customer and Employee. (The Customer and Employee forms don't exist in the sample database.) We'll then reference the following categorized view to bring the appropriate help data into each form:



When the form is sent to the browser, we load the help topics associated with that form into a set of JavaScript arrays. We can then manipulate those arrays on-the-fly to display the appropriate line of help text or pop-up text in the help frame or pop-up window. Before we discuss how to get this data from the view into a set of arrays, let's look at how this type of array works in JavaScript.

JavaScript arrays are extremely flexible. You can define them as named arrays, in addition to the more traditional simple integer arrays. If we were to *hard-code* the definition and population of the data you see in the above view, we might do it with the following code. Notice that the HelpLines array holds the single-line help text, while the HelpPopups array holds the pop-up text that is shown when the user clicks on the *More Info* link.

```
var HelpLines = new Array;
HelpLines["Email"] = "Please enter a valid Internet e-mail address";
HelpLines["SellName"] = "Check this box if you want to allow us release your name";
HelpLines["Phone"] = "Area code first, like this:  (317) 555-1212.";

var HelpPopups = new Array;
HelpPopups["Email"] = "We will <b>not</b> give this address out to anyone else, but will use it to contact you about the status of your order.";
HelpPopups["SellName"] = "This will allow us to release your name to other firms selling products we believe you would be interested in hearing about.";
HelpPopups["Phone"] = "Please include <b>country code</b> (if outside the United States) and an <b>extension</b> if applicable.";
```

The arrays contain the help text for the three reusable help topics on the Register form. Of course, we don't have to hard-code the JavaScript; we are going to let Domino generate it. But first, assuming we have coded the arrays as shown above, let's look at how the FrameHelpKey function uses the data defined in those arrays.

Here again is the HTMLAttributes formula for the Phone field:

```
HelpKey := "Phone";
"onFocus=\'FrameHelpKey(\"" + HelpKey + "\")\'"
```

Domino uses that formula along with the field definition to generate the field in HTML format, like this:

```
<INPUT NAME="Phone" onFocus='FrameHelpKey("Phone")'>
```

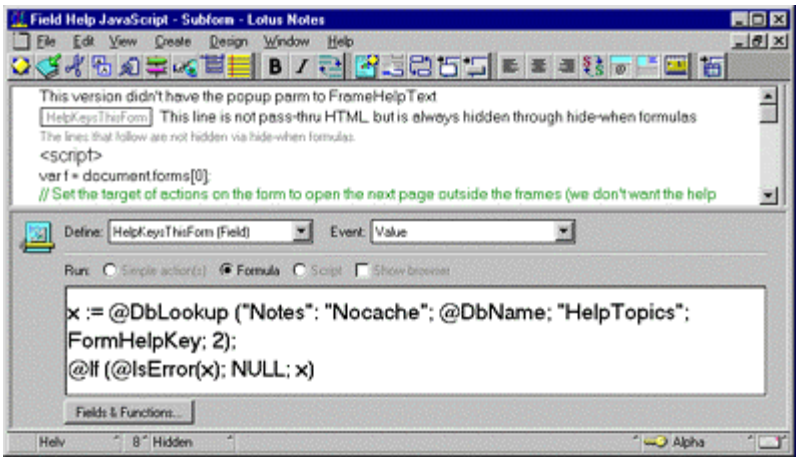
The FrameHelpKey function uses the Phone parameter (passed in as an argument named *key*) to write the appropriate line of help text from the HelpLines array to the help frame (only the relevant portion of the FrameHelpKey function is shown here):

```
function FrameHelpKey(key) {
  window.parent.HelpFrame.document.write("<Center><font size=4 color=green>" + HelpLines[key]);
  // Pull the message from the HelpPopups array, and generate it as a hard-coded parameter in the showHelp call on
  // the popup window.
 if (HelpPopups[key] == "") {
    window.parent.HelpFrame.document.write("</font></Center>");
    // Close out the formatting tags since there is no More Info link to be shown
  } else {
  window.parent.HelpFrame.document.write (" -- ");
  window.parent.HelpFrame.document.write("<A HREF='JavaScript:top.MainFrame.showHelp(\"" + HelpPopups[key]
+ "\")'>");
  window.parent.HelpFrame.document.write("More Info</A></Center></font>");
  window.parent.HelpFrame.document.close();
  }
}
```

This function also passes the appropriate pop-up help text from the HelpPopups array to the showHelp function, which creates the pop-up window.

Let Domino generate the code for you

Now back to the real challenge: dynamically loading the arrays with data from the HelpTopics view, instead of hard-coding each element in the array. We load the arrays using a combination of Domino and JavaScript. If you look at the "Field Help JavaScript" subform, just above the <script> tag, you'll see a computed-for-display field called *HelpKeysThisForm:*



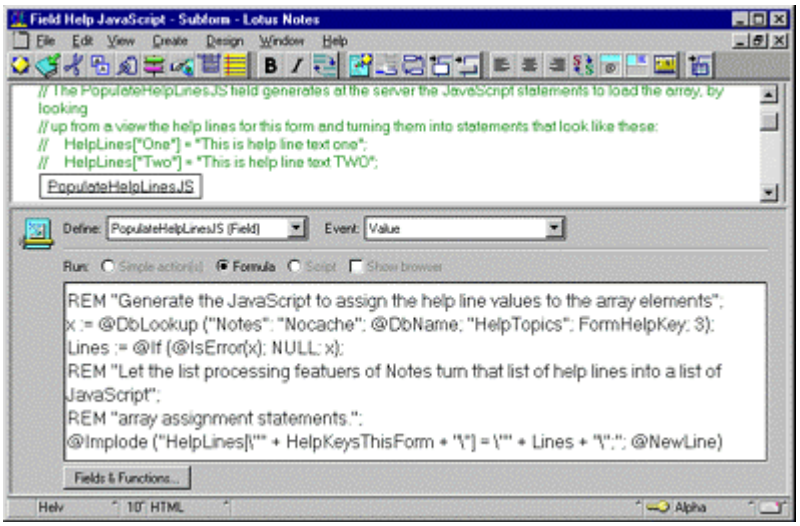
This field makes the list of help keys for the current form -- in this case Email, SellName, and Phone -- available in a Notes computed-for-display field for use by other computed field formulas.

Notice that we didn't say that this computed-for-display field makes those values available for use in our JavaScript code. It doesn't, at least not directly. Remember, JavaScript has access to fields defined on an HTML form, and computed-for-display fields generate what the browser sees as static text. If a hide-when formula is enabled, as it is here, the data isn't sent to the browser at all. However, it is available to other Notes field formulas on the form.

So, we have this computed-for-display field called *HelpKeysThisForm*, which calculates the help keys used. We can use a second computed-for-display field within the JavaScript code itself, a formula that references the keys calculated by HelpKeysThisForm to generate the code required to populate an array. Remember, the data calculated in a computed-for-display field is sent to the browser as static text for the browser to display or interpret. If that "static text" happens to be between the tags <script> and </script>, the browser's JavaScript interpreter attempts to parse it. So, all we need to do is calculate, in a Notes formula, the JavaScript code with the correct syntax to populate the HelpLines array! Remember, this is the JavaScript we want to have Domino generate:

```
var HelpLines = new Array;
HelpLines["Email"] = "Please enter a valid Internet e-mail address";
HelpLines["SellName"] = "Check this box if you want to allow us release your name";
HelpLines["Phone"] = "Area code first, like this:  (317) 555-1212.";
```

Actually, we'll hard-code the declaration of the array, and call the computed-for-display field *PopulateHelpLinesJS*:



Here is the formula for the PopulateHelpLinesJS computed-for-display field:

```
REM "Look up the help lines for this form";
x := @DbLookup ("Notes": "Nocache"; @DbName; "HelpTopics"; Form; 3);
Lines := @If (@IsError(x); NULL; x);
REM "Let the list processing features of Notes turn that list of help lines into a list of JavaScript";
REM "array assignment statements.";
@Implode ("HelpLines[\"" + HelpKeysThisForm + "\"] = \"" + Lines + "\";"; @NewLine)
```

The *Lines* variable is loaded with a text list -- each item contains a help line. Since the values load from the same view as the HelpKeysThisForm computed-for-display field (which was calculated to *Email : SellName : Phone*), they correspond element by element. And the last line of this formula (beginning with @Implode) combines the help keys with the help lines to produce the statements we've discussed.

Now, that's letting Domino do the work!! We didn't have to write a loop to go through each value -- the formula language automatically creates a list of items from the list of keys and help lines we've concatenated. If you open the Register form in a browser and then view the source, you'll see those lines of JavaScript.

It may seem strange to include a computed-for-display field right on a Notes form in the middle of a bunch of pass-thru JavaScript code. It helps to think about the way Domino and the browser interact. Domino generates an HTML page that it sends to the browser. Any area of a Notes form marked as pass-thru HTML is not formatted for the browser by Domino. In other words, if you have bold text, Domino won't generate an HTML bold tag -- <b>. This way, you can code your own HTML tags. But Domino will still compute any field formulas, including computed-for-display formulas. So, we use computed-for-display formulas to generate portions of HTML and/or JavaScript. That HTML stream, which includes all the JavaScript code, is then sent to the browser for processing. So, the browser acts as if we'd hard-coded every piece of HTML and JavaScript.

We load a second array, HelpPopups, using this same technique. It contains the text to be shown in the pop-up window for each field. Here is the formula for the PopulateHelpPopupsJS computed-for-display field, which generates the array assignment statements:

```
x := @DbLookup ("Notes": "Nocache"; @DbName; "HelpTopics"; Form; 4);
Popups := @If (@IsError(x); NULL; x);
@Implode ("HelpPopups[\"" + HelpKeysThisForm + "\"] = \"" + Popups + "\";"; @newline)
```

And here is the JavaScript it generates:

```
HelpPopups["Email"] = "We will <b>not</b> give this address out to anyone else, but will use it to contact you about the status of your order.";
HelpPopups["SellName"] = "This will allow us to release your name to other firms selling products we believe you would be interested in hearing about.";
HelpPopups["Phone"] = "Please include <b>country code</b> (if outside the United States) and an <b>extension</b> if applicable.";
```

Now we've seen how to generate the JavaScript to load the arrays, and how to pull data from those arrays dynamically, at the browser, to display text in the help frame and in the pop-up window.

Now for the real world: The RegisterGold form

If you followed the instructions in the sidebar "**Quick steps to adding field help to your applications**," you used the RegisterGold form rather than the Register form. The RegisterGold form references the "Field Help Gold" subform, which has a modified version of the JavaScript functions that we've been looking at so far. We have two versions because we discovered several interesting problems as we developed the techniques. Rather than complicate the explanations of the basic techniques, we decided to cover them now in a separate section, and to put the more robust

solution in the Gold version of the form and subforms. (Note: We won't cover everything that's different on the RegisterGold form, but it is all documented on the form.)

The following issues are addressed in the Gold version of the solution:

- Making the no-hassle frames (described in the **first article**) work not just when a document is composed -- with an ?OpenForm -- but also when a document is re-opened in edit mode. Spawning a second frameset using *?OpenForm&Frame=0* doesn't work in the second case.
- The Netscape 4.x tabbing problem, as described earlier in the article. The Gold version determines the user's browser type, and then uses the appropriate code for that browser type. This issue is what adds most of the complexity to the Gold solution.

Making no-hassle frames work for documents opened in edit mode

First, we want to get our field help to display in the separate, "no-hassle" help frame when a document is opened in edit mode. To do this, we can modify the $$HTMLHead formula to use the CGI variable *Path_Info,* which contains the current URL. Here is the modified formula for the $$HTMLHead formula on the RegisterGold form, with the changes in bold:

```
REM "compute the full path to this database (replacing backslashes with front slashes, and spaces with plus signs), for use below in URLs";
db := @ReplaceSubstring(@Subset(@DbName;-1);"\\" : " ";"/" : "+");
@If (
  @Contains (Query_String; "Frame=0") | @Contains (Query_String; "OpenDocument");
    @Return (NULL);
NULL
);
REM "If the a form name was used on the URL we will need to append ?OpenForm because we are also";
REM "appending other parameters using the & symbol";
PathToUse :=
  @If (
    @Contains (Path_Info; "?Open") | @Contains (Path_Info; "?Edit") ;
       Path_Info;
     Path_Info + "?OpenForm"
);
"<html><head><TITLE>" + FormTitle + "</TITLE>  </head> <frameset rows=\"90%,*\" frameborder=yes> <frame name=MainFrame scrolling=vertical src=" + PathToUse + "&Frame=0> <frame scrolling=no name=HelpFrame src=/"+db+"/FieldHelpGold?ReadForm&HelpFrameDefault=" + HelpFrameDefault + "> </frameset> <body> </body> </html>"
```

This formula also accounts for one fairly common scenario. When a user is creating a document, the URL should end in *?OpenForm.* If the user edits an existing document, the URL should end in *?EditDocument*. Either way, we want to append *&Frame=0* to that URL to re-open the RegisterGold form in the top frame. But we cannot simply append *&Frame=0* to what's found in *Path_Info*, because your application might have URLs that have been coded without the *?OpenForm* on the end (this is a shortcut you may use yourself: if there is no other design element called RegisterGold, Domino assumes a URL ending in RegisterGold includes an implicit *?OpenForm*). So, the following URL is valid, because Domino assumes the *?OpenForm* on the end:

http://hostname/dbname/**RegisterGold**

However, if you want to append parameters with an & symbol (such as, *&Frame*= to define frames), Domino requires you to explicitly use the *?OpenForm*. So, the following URL is not valid:

http://hostname/dbname/**RegisterGold&Frame=0**

Instead, you must use a URL like this one:

http://hostname/dbname/**RegisterGold?OpenForm&Frame=0**

So, our revised $$HTMLHead formula takes care of appending the *?OpenForm* if it is not already there.

Detecting the browser type and version
As discussed earlier in the article, certain types of JavaScript functions on a form cause Netscape 4.x browsers to stop tabbing properly between fields. So, we want to detect the type of browser that the user is using, and then display the appropriate help. Detecting the browser type is easy with JavaScript. A simple *if (ver == "nn3" || ver == "nn4" || ver == "ie301" || ver == "ie302" || ...* will tell you. (For more information on detecting browser types with JavaScript, see the article **Domino and JavaScript: Dynamic Partners (Part II)** article.)

But, we can't rely on JavaScript to tell us the browser type for two reasons. First, we don't want any JavaScript behind the field events for Netscape 4.x browsers, because the JavaScript is what causes the tabbing problem to occur in the first place. Second, if the browser version is Netscape 4.x, we want to use an anchor tag (hot link) for a field label with the HTML construct *STYLE="text-decoration: none"*. This makes the field labels hot (able to have an *OnMouseover* JavaScript event associated with them) without having them underlined like a traditional HTML link. That way, we can instruct the user to point to a field label for help, but we don't have to wreck the look of our form with underlined field labels. This HTML construct *text-decoration: nonestyle* is not supported in the older browsers.

Another solution is to use the CGI variable *HTTP_USER_AGENT* to tell us the browser type. Unfortunately, it returns values like these:

Mozilla/2.0 (compatible; MSIE 3.02; Windows NT)
Mozilla/2.0 (compatible; MSIE 3.03; Windows 3.1)
Mozilla/3.01 Gold (Win95; I)
Mozilla/3.0 (Win95; I)

The first two are Internet Explorer 3.x browsers, while the second two are Netscape 3.x browsers. But it's certainly not easy to tell which is which! And there are many possible values. For example, in a Web site we work with in which 375 Lotus, Microsoft and Novell certified professionals registered, we detected 108 different values in this field!

So, our solution on the RegisterGold form is to first capture the CGI variable in the computed-for-display field, *HTTP_User_Agent*. Then, we use a computed field called *Browser* immediately after the HTTP_User_Agent field to tell us the browser type. The Browser field has this formula:

```
@If (
  @Contains (RegistrationBrowser; "MSIE 4");
    "IE4";
  @Contains (RegistrationBrowser; "Mozilla/4");
    "Netscape4";
  @Contains (RegistrationBrowser; "MSIE 3");
    "IE3";
  @Contains (RegistrationBrowser; "Mozilla/3");
    "Netscape3";
 "Unknown: " + RegistrationBrowser
)
```

This works for most browsers, telling us which browser is present.

The *Label Hover* solution versus the *Field Focus* solution
As we described in the **first article**, we tweaked the basic field help technique for Netscape 4.x browsers. We still use the same JavaScript functions to display the text in the help frame (FrameHelpText and FrameHelpKey) and in

the pop-up window (showHelp). But we use them differently. Since having the *onFocus=FrameHelpText...* call in each field's HTML attributes (what we call the "Field Focus" solution) causes a tabbing problem with Netscape 4.x, we removed that call for Netscape 4.x browsers. Instead, we make the field label an anchor tag (we call this the "Label Hover" solution) using the following type of call:

```
<A STYLE="text-decoration: none" onMouseOver="FrameHelpText('Please enter your full name.', '', 'LabelHover')"
onMouseOut="FrameHelpText('', '')"  HREF="JavaScript:FrameHelpText('Please enter your full name.', '',
'LabelHover')">Name:</A>
```

This is the native HTML that our technique generates for the Name field's label; it sends that tag to the browser instead of a static text label *Name,* but only if the browser is Netscape 4.x. When the mouse moves over the anchor tag, the help text displays. And, when the mouse moves away from the anchor tag, the help frame clears. Since there is no pop-up help for this field, the *HREF* specified just re-displays the help text in case the user clicks on the label.

We don't hard-code this HTML, of course. What we do is have a computed-for-display field in place of the static text label that tells Domino to generate the above HTML for Netscape 4.x browsers. Or better yet, we code it so that you can decide to use the *Label Hover* help for both Internet Explorer 4.x and Netscape 4.x browsers, if you prefer (since the hot-but-not-underlined links don't work in 3.x browsers, you wouldn't want to use this solution for all browsers).

What we've done, then, is add two indicator fields, defined as computed-for-display fields after the Browser field. One is called *LabelHoverHelp* and the other is called *FieldFocusHelp.* Here is the formula for LabelHoverHelp:

```
@If (Browser = "Netscape4";  1; 0)
```

You can easily change this to include Internet Explorer 4.x (IE4) as well.

Next, we have a computed-for-display field in place of each static text label. The formula generates the anchor tag HTML shown above if LabelHoverHelp is true, or simply a static text label otherwise. Here is the formula for the Name field label, which generates the HTML anchor tag we described above:

```
LabelText := "Name:";
msg := "Please enter your full name.";
popup := NULL;
type := "LabelHover";
href :=
@If (popup = NULL;
    "JavaScript:FrameHelpText(\'" + msg + "\', \'" + popup + "\', \'" + type + "\')";
"JavaScript:showHelp(\'" + popup + "\')"
);
@If (
  @IsMember ("$$WebClient"; @UserRoles) & LabelHoverHelp;
onMouseOver=\"FrameHelpText(\'" + msg + "\', \'" + popup + "\', \'" + type + "\')\" onMouseOut=\"FrameHelpText(\'\',
\'\',\'\')\"  HREF=\"" + href + "\">" + LabelText
;
LabelText
)
```

If LabelHoverHelp evaluates to true, the anchor tag displays. Otherwise, the static label text displays. Notice that the anchor tag also contains not only the *onMouseOver* and *onMouseOut*, but also the *HREF* to display the pop-up help. So, when the mouse moves over the label, the help frame text appears. When the mouse moves away from the label, the help text disappears. Clicking on the label causes the pop-up help to appear.

You'll notice that the FrameHelpText function has an extra parameter now, in addition to the help text and pop-up text: a *type* parameter. In this case, we pass it the value *LabelHover*. The FrameHelpText function on the "Field Help JavaScript Gold" subform looks like this:

```
function FrameHelpText(msg, popupText, HoverOrFocus) {
  // Write the help text to the help frame
```

```
  window.parent.HelpFrame.document.write("<font size=4 color=green><Center>" + msg);
  if (popupText == "") {
    window.parent.HelpFrame.document.write("</font></Center>");
    // Close out the formatting tags since there is no More Info link to be shown
  } else {
  if (HoverOrFocus == 'FieldFocus') {
      // Show the More Info link
      window.parent.HelpFrame.document.write (" -- ");
      window.parent.HelpFrame.document.write("<A HREF='JavaScript:window.parent.MainFrame.showHelp(\"" +
popupText + "\")'>");
      window.parent.HelpFrame.document.write("More Info</A></Center></font>");
    } else {
        // Assume 'LabelHover' -- they can just click on the field label
        window.parent.HelpFrame.document.write (" -- ");
        window.parent.HelpFrame.document.write("<font size=3 color=red>Click on the label for more
information</font></center>");
        window.parent.HelpFrame.document.close();
    }
  }
  window.parent.HelpFrame.document.close();   // this closes out the document (allowing the text to display), not the
frame or window
}
```

The "FieldFocus" approach works the way we described earlier in the article: the text is written to the help frame along with a hot link to display the pop-up text. It's the "LabelHover" approach that is new. We don't want the *More Info* link to display when the Netscape 4.x user hovers over (points to) the field label, because when the user moves the mouse away from the field label toward the link in the help frame, the *onMouseOut* event would make the help frame text disappear (it disappears so that it's not left there confusing the user when the mouse is somewhere else on the page, but it also makes the *More Info* link disappear when you try to click on it). So, instead of a *More Info* link, we tell the user to "click on the label for more information." And the HREF for the field label anchor tag contains the call to the showHelp routine.

Conclusion

Now you know how to put readable, context-sensitive field help into your Domino applications, using techniques that work together to handle all the major browsers in use today. Readable field help appears in a separate frame when most users tab into a field, but appears instead for Netscape 4.x users when they point to a field label. Your users will get help that's appropriate to the form they're filling out. And since you can let the non-technical owners of your application maintain reusable help text via Notes or a browser, you should find your Domino Web applications are truly easier to use!

ABOUT MICAH

Micah Blalock is an Internet consultant and instructor at WorkFlow Designs, Inc. in Dallas, Texas. Blalock has wide experience developing solutions for the commercial database, medical and PC game industries. Currently, his focus is on development and technical training with specialization in application development for Internet technologies such as Lotus Notes/Domino and JavaScript. Blalock also lends his combination of technical and training expertise in developing the technique-oriented courseware for Internet development offered through WorkFlow Designs TopGun Academy curriculum.