Lotus. Developer Domain | The Technical Resource for Lotus software | LDD Today

Enhancements to the formula language in Domino 6

Level: Advanced
Works with: Domino 6
Updated: 04-Nov-2002

by Michelle Mahoney
and Robert Perron

The introduction of programming languages into Notes/Domino, starting with LotusScript in R4, led to forecasts of the demise of the formula language. But the formula language remains a favored tool because its source code is compact and its performance excels. The fly in the ointment has been the lack of control features found in programming languages, particularly loops.

Lotus Notes/Domino 6 overhauls the formula language and introduces many new @functions. This article concentrates on the following enhancements to the formula language control features:

- Temporary variables can be reused.
- Lists can be subscripted.
- Assignment statements can be nested.
- Braces can be used to delimit remarks and constants.
- Documents can be locked programmatically.
- The context of a formula can be changed within a single statement.
- Loops become possible with @For, @While, and @DoWhile.
- @IfError, @Eval, and @CheckFormulaSyntax enhance error processing.
- List processing enhancements include @Compare, @Transform, @Nothing, @Count, @Sort,@Max (one parameter), and @Min (one parameter).
- @GetField, @ThisName, and @ThisValue let you access fields without having to hardcode their names.
- New functions, such as @WebDbName, @URLEncode, and @URLDecode, make life easier for Web developers.
- You can quickly retrieve LDAP listener information and convert Notes-formatted names to LDAP-formatted names and vice versa.

This article is intended for Notes application developers with formula language experience.

Pre-Notes/Domino 6 iteration

The pre-Notes/Domino 6 formula language provides two powerful, but limited, techniques for processing lists:

- Operators and @functions can perform the same operation on every element of a list or lists. For example, with a single statement, you can add the corresponding elements of two lists, or you can sum the elements of one list.
- Agents can run one formula on multiple documents. For example, with a single statement, you can add two fields to every document in a view.

These capabilities allow you to do a lot of work with very little coding. They are akin to vector instructions in vector-processing machines.

However, they have a significant shortfall in that, like vector instructions, they do not allow operations that have dependencies on other operations in the list. Suppose you are operating on the corresponding elements of two lists, and each operation needs the result of the operation on the preceding element in the list. Or suppose you want to get input for each list element with @Prompt. In these cases, you would have to serialize the operations.

This is where list processing falls short.

The following code illustrates a simple list operation. You might use it in a button on a form.

```
REM {This single statement assigns all the elements of one list to another.};
FIELD ListResult := ListOne
```

Here's another:

```
REM {This statement adds the corresponding elements in two lists.};
FIELD ListResult := ListOne + ListTwo
```

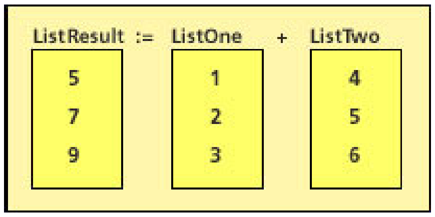
And here's a reduction operation:

```
REM {This statement sums the elements in one list.};
FIELD Result := @Sum(ListOne)
```

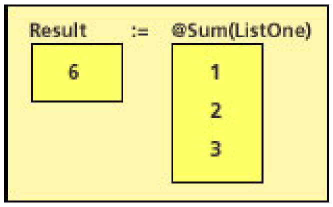
These list operations work in pre-Notes/Domino 6 as well as Notes/Domino 6 formulas.

Note: Notice the use of braces to delimit remarks in the examples. This is new in Notes/Domino 6 and is especially handy for delimiting remarks containing quotation marks. For more about the use of braces in Notes/Domino 6, see the **Braces** sidebar.

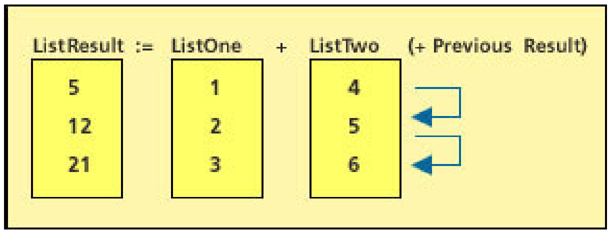
The problems illustrated in the previous examples can be expressed as list operations. For example, in the second problem, each constituent operation stands by itself, as illustrated below:



In the third example, the constituent operations have no dependencies; you can sum them in any order, or, for example, you could sum them in groups, and then sum the groups:



However, there is another problem for which we have not provided a solution yet: each constituent operation depends on the result from the preceding constituent operation. The list elements must be processed sequentially:



Locking documents

One of the strengths of Notes/Domino 6 is the ease with which developers can now share access to design elements, databases, and documents with other team members. To preclude any design conflicts that can arise when different developers simultaneously access the same design element, this feature also incorporates a mechanism for locking these elements. The same functionality has been extended to Notes documents. You can now both share and lock documents to manage the access that team members have to a document's data. Locking documents also prevents save and replication conflicts.

Only documents stored in a database that is set to allow document locking can be locked. To enable locking, the manager of the database must designate a server to be the administration server, which is also referred to as the *master lock server,* in the database's Access Control List. Once a document is locked only the lock holder and the database manager can make changes to the document, regardless of whose name exists in a Readers or Authors field on the document. And only the user who locked the document or the database manager can unlock it.

@DocLock's keyword options enable you to find out if locking is enabled for the database([LockingEnabled]), to determine the locked status of the current document([Status]), and to lock ([Lock]) or unlock ([Unlock]) the current document. For example, the following code, when stored in the Lock hotspot button and triggered on a previously saved document, does one of the following:

- If document locking is not enabled for the database, a message box appears stating "Locking is not enabled for this database."
- If document locking is enabled, but the current document is already locked by another user, a message box appears stating "Document is already locked by <user's hierarchical name>."
- If document locking is enabled and the current document is not locked, document locking locks the document. The current user is designated as the owner of the lock by the administration server.

```
@If(@IsNewDoc;@Prompt([OK];"Save first";"You can't lock a new document; save, close, and reopen the
document first.");@If(@DocLock([LockingEnabled]) = 1;@If(@DocLock([Status]) =
"";@DocLock([Lock]);@Prompt([OK];"Already locked";"Document is already locked by " +
@DocLock([Status])));@Prompt([OK];"Not enabled";"Locking is not enabled for this database")))
```

The following code, when triggered from the Unlock hotspot button on a previously locked document either unlocks the document (if it is locked by the current user) or displays the name of the current lock owner:

```
@If(@IsNotMember(@V3Username;@DocLock([Status]));@Prompt([OK];"Not authorized";"Document was locked
by " + @DocLock([Status]) + ". Contact the lock owner");@DocLock([Unlock]))
```

@UpdateFormulaContext

Traditionally, formulas have been able to manipulate only the data stored in the form or view in which they were written. With Notes/Domino 6, you can now change the context of the code within the formula to manipulate data in documents or views outside the current design element. For instance, you can write a formula that first opens a form to create a new document. Then, from within the same formula, you can add data to this new document. Simply adding @UpdateFormulaContext to the formula changes the context of the code to apply to the newly created or opened design element.

Here's an example of @UpdateFormulaContext at work. The following code creates a new document, copies the user name values in the fname and lname fields from the current document, then adds them to two fields of the same name in the newly created document—all in one formula.

```
tempfname := fname;
templname := lname;
@Command([Compose];"userinfo");
@UpdateFormulaContext;
@SetField("fname";tempfname);
@SetField("lname";templname)
```

This same formula in pre-Notes/Domino 6 applications merely reassigns the name field values of the current document. Once you add @UpdateFormulaContext to the formula, the field names referenced after it are treated as fields on the new document, not the existing document.

You can also use @UpdateFormulaContext to change a value in a document in a view based on the content of another document in that view. For example, if you are a runner, you might keep a running log in which you track the distance and speed of each run as a separate view entry. If, for instance, you decide you also would like to track how much your speed is improving by comparing speeds from one run to the next, you could add the following code to an action button for the view. This code retrieves the value of the speed field from the previous document in a chronological view and subtracts it from the value of the speed field on the currently highlighted document, then places the result (which is hopefully a positive number) in the "improved" field on the highlighted document:

```
@Command([NavPrev]);
@Command([EditDocument]);
@UpdateFormulaContext;
valuetopass := speed;
@Command([NavNext]);
@UpdateFormulaContext;
@SetField("improved";(speed - valuetopass));
@Command([FileSave])
```

@For loops in Notes/Domino 6

To understand how you use loops in Notes/Domino 6, let's start off with a simple loop that assigns ListOne to ListResult. This does the same thing as ListResult := ListOne. In practice, you'd use the list operation; this exercise is just to analyze the loop and go over the new features.

```
REM {Assign ListOne to ListResult using an @For loop.
Use concatenation on all but the first iteration.};
@For(n := 1; n <= @Elements(ListOne); n := n + 1;
    FIELD ListResult := @If(
        n = 1;
        ListOne[n];
        ListResult : (ListOne[n])
    )
)
```

This short piece of code contains many new features:

- The temporary variable *n* is reassigned over and over! The inability to reassign temporary variables in pre-Notes/Domino 6, except with @Set, made doing anything with a variable a number of times within a formula all but impossible.
- ListOne is subscripted! You can refer to a single element of a list with an integer subscript, where 1 is the first element. In pre-Notes/Domino 6 formulas, working with individual list elements was difficult at best. @Implode and @Explode provided some rudimentary capabilities, but not what was needed to loop through lists. Use @Elements to determine the size of the list.
- This loop uses the new @For, which is, predictably, similar to For loops in programming languages. The first parameter assigns the beginning iteration to the loop variable, the second parameter specifies the condition for each iteration, the third parameter increments the loop variable, and the remaining parameters comprise the body of the loop.

One restriction in using subscripts is that they cannot appear on the left-hand side of an assignment. So in the previous code example, you cannot do:

```
ListResult[n] := ListOne[n]
```

You have to build ListResult by concatenating the ListOne elements.

As usual, be sure to use parentheses around expressions being concatenated because the concatenation operator has a high precedence.

More simple loops

The next loop duplicates the second list processing example (ListOne + ListTwo):

```
REM {This code adds two lists using a loop.};
@For(n := 1; n <= @Elements(ListOne); n := n + 1;
    FIELD ListResult := @If(
        n = 1;
        ListOne[n] + ListTwo[n];
        ListResult : (ListOne[n] + ListTwo[n])
    )
)
```

The following loop sums the elements in one list (@Sum of ListOne):

```
REM {This statement sums the elements in one list using a loop.};
```

```
FIELD Result := 0;
@For(n :=1; n <= @Elements(ListOne); n := n + 1;
    FIELD Result := Result + ListOne[n]
)
```

The problem loop
Now we get to the problem that you cannot resolve with list operations. For each set of elements in two lists, we want to add them and also add the total from the preceding list element. (Another way to look at the problem is that for each set of elements, we wish to sum all the elements to that point.) Each constituent operation needs the result from the previous constituent operation.

```
REM {We cannot do this with a list operation because each add
uses the result of the previous add.};
@For(n := 1; n <= @Elements(ListOne); n := n + 1;
    FIELD ListResult := @If(
        n = 1;
        ListOne[n] + ListTwo[n];
        ListResult : (ListResult[n-1] + ListOne[n] + ListTwo[n])
    )
)
```

@While loops
Here's some code using the new @While. It's longer than it has to be so that you can see what's going on. @While checks the condition specified by the first parameter at the beginning of the loop. If the condition is true, the body of the loop (the remaining parameters) executes. Then we loop back to the condition.

```
REM {We cannot do this with one operation because the @Prompt statements must be serial. This loop collects
string items one item at a time by prompting the user and adds them to a list.};
item := @Prompt([OKCANCELEDIT]; "Item"; "Enter item or \" \""; "");
@While(
    item != "";
    FIELD ListResult := @If(
        ListResult = "";
        item;
        ListResult : item
    );
    item := @Prompt([OKCANCELEDIT]; "Item"; "Enter item or \" \" to quit"; "")
)
```

The example gets user input with @Prompt. @While processes the body of the loop if the user input is not an empty string. (The user can also exit by pressing Cancel in @Prompt.) The code then concatenates the item to ListResult or assigns it if ListResult is empty. Finally, the code prompts again and loops back to the test.

Below is a condensed version of the same code. Instead of assigning item before the loop and as the last statement of the loop body, we assign it once as part of the @While conditional statement. Notice that the assign statement is nested.

```
REM {This is a condensed version of the @While loop.};
@While(
    (item := @Prompt([OKCANCELEDIT]; "Item"; "Enter item or \" \""; "")) != "";
    FIELD ListResult := @If(
        ListResult = "";
        item;
        ListResult : item
    )
)
```

Here is an @DoWhile loop. It displays elements in a list one at a time from the first element through either the last element or the element whose content is END*OF*LIST. The conditional statement comes at the end.

```
REM {An @DoWhile loop processes the loop body
before checking the condition.};
@If(ListOne = ""; @Return(0); "");
```

```
n := 0;
@DoWhile(
	n := n + 1;
	@Prompt([OK]; "Element " + @Text(n); ListOne[n]);
	((n < @Elements(ListOne)) & (ListOne[n] != "END*OF*LIST"))
)
```

Enhancements to @Max and @Min

Previously, you could use @Max and @Min to determine the largest or smallest of two numbers or to produce a number list of the largest or smallest numbers resulting from a pair-wise comparison of two number lists. With Notes/Domino 6, you can also pass @Max and @Min a single number list to determine the largest or smallest number in that list. For instance, in R5 you could use @Max to determine which sales person performed the best in overall sales in Q1:

```
maria_sales := 150000:80000:50000;
anthony_sales := 25000:170000:20000;
best := @Text(@Max(@Sum(maria_sales);@Sum(anthony_sales)));
@If(best= @Text(@Sum(maria_sales));"Maria with " + best;"Anthony with " + best)
```

The result of this formula is Maria with 280,000. You could also list the three worst monthly sales totals in the quarter:

```
@Min(maria_sales;anthony_sales)
```

The result of this formula is 25,000; 80,000; 20,000. With Notes/Domino 6, @Max can now determine which month's sales total was the highest overall:

```
maria := @Max(maria_sales);
anthony := @Max(anthony_sales);
@Max(maria;anthony)
```

The result of this formula is 170,000, Anthony's mid-quarter monthly sales total. The ability to extract the largest or smallest value in a single number list as a single value is a useful enhancement to @Max and @Min.

@Compare

Not only have existing functions been extended to improve list manipulation, Notes/Domino 6 also introduces new functions to serve this purpose. @Compare is one of these; it adds flexibility to handling text lists in particular. @Compare is similar to @Max or @Min in that it lets you compare two lists and identify whether an element in the second list is greater than or less than the corresponding element in the first list. The difference is that @Compare compares text or text lists instead of numbers or number lists. It derives the value of a character in a text list based on its position in the alphabet. Using @Compare, you can now extract all the words that begin with *a*, *b*, or *c* from a text list in field custName, for instance:

```
@For(n := 1; n <= @Elements(custName); n := n+1;
FIELD result := @If(n = 1;@If(@Compare(custName[n];"d"; [CaseInsensitive]) =
-1;custName[n];"");@If(@Compare(custName[n];"d"; [CaseInsensitive]) = -1;result:(custName[n]);result)));
result
```

@Sort

@Sort is another new function that adds flexibility to handling lists; it can be used with any type of list: text, number, or date-time. With @Sort, you can arrange the order in which elements are displayed in a list. In the following example, you can retrieve the prices of a product from the Price column (which is the fourth column in the productView view) for a specific product family and display them in descending order:

```
@Sort(@DBLookup("";"Server/Name/Notes":"Directory\\DbName.nsf";
"productView";"ProductFamilyKeyword";4);[Descending])
```

Or you can sort a text list to display in reverse alphabetical order. For instance, you can use the following formula if the movies field contains the list: titanic : Jaws : Wizard of Oz:

```
@Sort(movies;[Descending]:[CaseInsensitive])
```

The formula returns Wizard of Oz; titanic; Jaws. Sort also has a powerful keyword option called [CustomSort], which reserves two variables, $A and $B, that you can use in a formula to compare the elements in a list two at a time. For example, you can use @Sort to display the movie list in order from the shortest to the longest movie title:

```
@Sort(movies;[CaseSensitive]:[CustomSort];@If(@Length($A) < @Length($B);-1;@Length($A) >
@Length($B);1;0))
```

This formula returns Jaws;titanic;Wizard of Oz.

Notice that the custom sort keyword overrides the case sensitivity keyword.

@Transform and @Nothing

An additional enhancement to list operations is provided by the new @Transform function. With @Transform, you can apply a formula to each element of a list and return a list that displays the result per element. For instance, you can use @Transform to ensure that names are always formatted with the first name followed by the last name, regardless of how a user enters a name into a field. Suppose the Names field contains the following text list:

```
"James Brown" : "White, Fred" : "Smith, John" : "Doug Smith"
```

Using the following formula transforms the list in the Names field to: James Brown : Fred White : John Smith : Doug Smith:

```
@Transform( Names; "x";
        @If(@Contains( x; ",");
                @Right( x; ",") + " " + @Left( x ; ",");
                x))
```

to return the list:

```
"James Brown" : "Fred White" : "John Smith" : "Doug Smith"
```

You can use @Nothing when you don't want to return a result during an interation. For example, if a field contained the results of the previous formula, and you wanted to capture only the names that start with J, you would use:

```
@Transform( Names; "x";
        @If( @Begins( x; "J");
                x;
                @Nothing));
```

This returns:

```
"James Brown" : "John Smith"
```

@Count

@Count limits the possibility of generating errors when you capture the number of elements in a text, number, or date-time list. It is identical to the @Elements function, except that @Count does not return a zero if the value supplied as a list is null or does not have a list data type; instead, it returns one. This prevents errors from being generated when the element count is used in conjunction with other functions to perform complicated tasks.

For instance, if you want to return the third entry in a view containing six entries, you could use the following code:

```
names := @DBLookup("";"Server/Name/Notes" : "names.nsf" ; "Groups"; "MyCoworkers" ; "Members");
@Subset(@Subset(names;@Elements(names)/2));-1)
```

However, if the lookup fails and returns no names, the error "The second argument to @Subset must not be zero" is displayed. To prevent this error, you can change @Elements in the formula above to @Count. This function instead returns the first entry in the retrieved list.

Error checking enhancements

Notes/Domino 6 formula language includes a number of new error checking functions.

@IfError
@IfError is a new function that enhances error handling because it lets you customize an error message to replace the generic error message generated by Notes.

For example, if you want to compare the monthly sales totals for Maria and Anthony for the year, but you know that Anthony took a leave of absence during the fourth quarter and you do not want his September sales total to be repeated three times when comparing the two lists (which is the default behavior), you could use the following formula:

```
tanthony := @Elements(anthony_yrtotal);
tmaria := @Elements(maria_yrtotal);
dif := (tanthony - tmaria);
result := @If(@Sign(dif) = -1;@Subset(maria_yrtotal; tmaria-@Abs(dif));
@Subset(anthony_yrtotal;(tanthony - dif)));
@If(@Sign(dif)=-1;@Max(anthony_yrtotal;result);
@Max(result;maria_yrtotal))
```

This formula displays the highest sales per month for the first nine months of the year; in other words, for only those months for which both salespeople posted results.

If none of Anthony's sales numbers were added to the form, leaving the anthony_yrtotal field null, this formula generates two errors, both triggered by the @Subset function. The @Subset function requires a list as its first parameter, and it cannot accept a zero as its second parameter.

To prevent the first error, you can inform the user that the anthony_yrtotal field is empty by rewriting the last line of the formula as follows:

```
@IfError(@If(@Sign(dif)=-1;@Max(anthony_yrtotal;result);
@Max(result;maria_yrtotal));"One of your list fields is empty");
```

Instead of displaying a cryptic error message, the @IfError function fills the target field with the text "One of your list fields is empty," which triggers the user to supply the missing data.

To prevent the second error, which reads "The second argument to @Subset must not be zero," replace the @Elements function in the tanthony and tmaria assignment statements with @Count. @Count returns a one instead of a zero when the field it is evaluating is null.

@Eval
@Eval enables you to execute a formula passed to it as a string at run-time. This means you can build formulas on the fly. For instance, you could enable users to execute formulas they write themselves into a dialog box. Just add the following code to a hotspot button on a form:

```
formula := @Prompt([OKCANCELEDIT];"Your formula";"Enter your formula
here";"@FunctionName(parameters)");@Eval(formula)
```

If the user enters the following code in the dialog box, his name displays in the Name field on the form.

```
@SetField("Name";@UserName)
```

Or, if the user does not know the names of fields on the form, he can save his name to the form by creating a field, assigning a value to it, and testing its contents using an @Prompt function:

```
FIELD myName := @Username;@Prompt([ok};"Contents of myName field";myName)
```

Alternatively, you can enable a user to change a value in a column in a view by creating an action button in the view that contains the same code as the previous hotspot button:

```
formula := @Prompt([OKCANCELEDIT];"Your formula";"Enter your formula
here";"@FunctionName(parameters)");@Eval(formula)
```

The user would have to include the @UpdateFormulaContext function in his formula to effect the desired change. If he enters the following code in the resulting dialog box, he can change the value of the color field of the currently selected document to blue.

```
@Command([EditDocument]);@UpdateFormulaContext;@SetField("color";"blue");@Command([FileSave]);@Com
mand([CloseWindow])
```

@CheckFormulaSyntax
@CheckFormulaSyntax checks a block of commented-out formula code for errors and reports the error message, line, column (number of characters from left to right in error line), offset (number of characters from left to right in formula block), and the text where the error is encountered. This function reports compile errors, not run-time errors. It finds errors that are normally caught by the Formula compiler and displayed in the error box at the bottom of the script area when you save the formula, but does so in code that is commented out and therefore, is ignored by the compiler.

A good time to use this function is when checking for errors in a formula entered by a user and evaluated outside the scope of the Formula compiler, as in our earlier example demonstrating @Eval. In the @Eval example, the user is entering text into a message box. Therefore, the compiler cannot check it for simple syntax errors. Here is an example of how you can use @CheckFormulaSyntax with @Eval:

```
formula := @Prompt([OKCANCELEDIT];"Enter a formula";"Add your formula
here";"@FunctionName(parameters)");
@If(@Implode(@CheckFormulaSyntax(formula) = "1";@Eval(formula);@Prompt([OK};"Checking the
syntax";@Implode(@CheckFormulaSyntax(formula))))
```

This code asks a user to enter a formula in a message box. It then checks the syntax of the formula entered by the user. If there are errors, it displays the error information and does not evaluate the formula. Otherwise, it evaluates the formula.

It is important to remember that when using @CheckFormulaSyntax, it returns compile errors, not run-time errors. So, @CheckFormulaSyntax returns "Unknown @Function 1116 @Propt" if the user mistypes @Prompt in the formula, as follows:

```
@Propt([ok];"Your name";@UserName)
```

However, it finds no errors and evaluates the formula if a user enters a formula with an insufficient number of arguments:

```
@Prompt([ok];"Your name")
```

The compiler accepts and evaluates this formula because the error it includes is a run-time error, not a compile error.

@GetField, @ThisName, and @ThisValue
@ThisValue and @ThisName let you write formulas that reference the current field without requiring you to hardcode the field name into the formula. This means that:

- Repeating code in multiple fields throughout an application is as easy as cutting and pasting it; you do not need to edit it to correct the field names.
- You no longer need to manually revise code when field names change, as they often can.

@ThisValue and @ThisName are especially useful for writing reusable translation and validation formulas. For instance, the following input validation formula requires a user to select more than one choice from the current list field. Normally, you would need to hardcode the field name into the formula, but with @ThisValue you can write:

```
@If((@ThisValue != "") & (@Elements(@ThisValue) = 1);@Failure("The " + @ThisName + "field must contain
more than one choice");@Success)
```

You can copy and paste this code to any other field in your application that requires the same validation test without having to edit it. The following input translation formula removes duplicate domains from the mail address for the recipient entered in the CopyTo field of a mail memo:

```
@OptimizeMailAddress(@ThisValue)
```

This same code can be used as-is in the SendTo and BCC fields as well.

@GetField provides an additional layer of functionality. When used with @ThisName, it makes it easier to create generic field formulas that you can reuse multiple times. For instance, if Maria and Anthony work for a doughnut retail chain, they might store their sales data in fields that look something like this:

ChocolateGlazed_quantity	ChocolateGlazed_price	ChocolateGlazed_total
HoneyDipped_quantity	HoneyDipped_price	HoneyDipped_total
Powdered_quantity	Powdered_price	Powdered_total

In R5, the ChocolateGlazed_total field consists of code similar to the following:

```
ChocolateGlazed_quantity * ChocolateGlazed_price
```

You use the same code for each total field, but edit it for each doughnut type. With Notes/Domino 6, you can use the following code in the ChocolateGlazed_total field:

```
doughnutName := @Left(@ThisName;"_");
(@GetField(doughnutName + "_quantity"))*(@GetField(doughnutName + "_price"))
```

With this code, you can keep the field name out of the code, so you can use the same code in the total field for every doughnut in the table.

Relief for Web developers

Goodbye @SubString. Notes/Domino 6 introduces a group of new functions that translate human-readable Notes titles and URL commands into Web-readable formats without requiring the use of @SubString to replace backslashes with forward slashes or worrying about whether the browser prefers plus signs over ampersands. Perhaps the longest anticipated of these new functions is @WebDbName, which returns the current database name converted to a URL-friendly format. It returns, for example, "Stuff/Customer%20List" if the current database name is called Customer List.nsf and resides in the Stuff subdirectory. What does this mean to you? It means that instead of writing:

```
@ReplaceSubstring(@Subset(@DbName;-1);"\\";"/")
```

you can write:

```
@WebDbName
```

Another new Web-savvy @function, @URLEncode, gives you the ability to determine the character set with which to encode a string into a URL-safe format. By default, the Domino 6 Web server uses UTF-8, which stands for UCS (Universal Character Set) Transformation Format 8. UTF-8 is an ASCII-compatible multi-byte Unicode and UCS encoding. Using @URLEncode, you can instruct the server to encode a string using a different character set, such as Shift_JIS, the character set that supports Japanese characters, or ISO-8859-1, which is an 8-bit, single byte coded graphic character set for European languages.

If you have a form entitled "Inglés," for example, and want to access it from a Web browser, you could write the following code in a hotspot button:

```
@URLOpen("http://MyServer/" + @WebDbName " + "/" + @URLEncode("Domino";"Inglés") + "?OpenForm")
```

Once the form opens, the resulting URL string looks like this:

```
http://MyServer.domain.com/Customers.nsf/Ingl%E9s?OpenForm
```

The %E9 is the UTF-8 encoding of the é in the form name. Because the function specified Domino, the server used Domino's default encoding character set, which is UTF-8.

To specify the use of the ISO-8859-1 character set instead, write the formula as follows.

```
@URLOpen("http://MyServer/" + @WebDbName" + "/" + @URLEncode("ISO-8859-1";"Inglés") + "?OpenForm")
```

The resulting URL string looks like this:

```
http://MyServer.domain.com/Customer.nsf/Ingl%C3%A9s?OpenForm
```

@UrlEncode gives you greater control over how your URL strings are interpreted by a Web browser. This function is very useful, especially for multi-language applications that are accessed from the Web.

@URLDecode turns strings in Web-ready format into human-readable strings. It again gives you the ability to specify the character set to use while decoding. You could use @URLDecode, for example, in an input validation formula for a field if the field contains an encoded URL.

A form with a field having the following default value formula:

```
"http://MyServer/"+@WebDbName + "/" + @URLEncode("Domino";"Inglés") + "?OpenForm"
```

displays the field as follows:

```
http://MyServer/Blank.nsf/Ingl%C3%A9s?OpenForm
```

If you add the following code as the input validation formula for the field:

```
@URLDecode("Domino";@ThisValue)
```

when the document is refreshed, the encoded URL is replaced with http://MyServer/Customers.nsf/Inglés?OpenForm, which is much easier for users to understand.

LDAP

LDAP (Lightweight Directory Access Protocol) is a protocol for accessing online directory services. Support for LDAP has been included in Domino since Release 4.6. The introduction in Notes/Domino 6 of the @LDAPServer function provides a quick means for accessing the IP address and port number of the LDAP listener for the current server. For instance, the following result is an example of what you might see if you enter @LDAPServer as the default value formula for a field, then access the form containing that field through a Web browser:

```
ldap://myServer.myDomain.com:389
```

In the result, myServer.myDomain.com is the IP address, and 389 is the port number of the LDAP listener.

Notes/Domino 6 also enhances the @Name function to enable conversion of Notes-formatted names into LDAP-formatted names and vice versa. The mapping between LDAP and Notes is as follows:

LDAP	Notes
ObjectClass	Form or subform
AttributeType	Field
Syntax	Data type

@Name now has the following keywords that handle this mapping for you:

- [TOFORM] returns the Domino form name when an LDAP ObjectClass name is provided.
- [TOOC] returns the LDAP ObjectClass name when a Domino form or subform name is provided.
- [TOAT] returns the LDAP AttributeType name when a Domino field name is provided.
- [TOFIELD] returns the Domino field name when an LDAP AttributeType name is provided.
- [TODATATYPE] returns the Domino data type name when an LDAP Syntax name is provided.
- [TOSYNTAX] returns the LDAP Syntax name when a Domino data type name is provided.

To get the LDAP AttributeType name for a person's assistant, for example, enter:

```
@Name([TOAT];"assistant")
```

The result is secretary, the LDAP term for an assistant. Or to find out the Domino term for the LDAP mail AttributeType, enter:

```
@Name([TOFIELD];"mail")
```

The result is Internet Address, which is also the name of the field containing a user's email address on the Person document in the Domino Directory. Likewise, entering @Name([TODATATYPE];"Integer") returns Number, which is the Notes term for the number data type and entering @Name([TOSYNTAX];"Text") returns Directory String, which is the LDAP term for the text data type.

Conclusion

In this article, we provided you with an overview of the changes to the Lotus formula language in Notes/Domino 6. To find out more about these changes, see the **Domino Designer 6 Help**.

ABOUT MICHELLE MAHONEY
Michelle Mahoney is a writer in the Notes/Domino User Assistance group and is currently working on updating and improving the formula language documentation for Notes/Domino 6.

ABOUT ROBERT PERRON
Robert Perron is a documentation architect with Lotus in Westford, Massachusetts. He has developed documentation for Lotus Notes and Domino for over six years with a primary concentration on programmability. He developed the documentation for the LotusScript and Java Notes classes and coauthored the book *60 Minute Guide to LotusScript 3 - Programming for Notes 4*. He authored the *Iris Today* article "**Common ground: COM access to Domino objects**."

Braces

In Notes/Domino 6, you use braces for delimiting remarks. You can still use quotation marks as delimiters when you enter the code; however, the next time you open the object containing the code, the delimiters will be generated as braces.

You can also use braces to delimit string constants. For example, you can enter:

```
Name := {"Bill" Jones}
```

Instead of:

```
Name := "\"Bill\" Jones"
```

In this case, when you next open the object containing the code, the delimiters will be generated as quotation marks, and embedded quotation marks will be escaped with backslashes—just the opposite of remarks.