Objectives

- Use constants in fields and formulas
- Use formulas in forms
- Use formulas in views
- Use formulas in agents
- Use formulas in actions
- Understand new and enhanced @Functions
- Understand new and enhanced @Commands
- Troubleshoot formulas

CHAPTER 4

Coding Formulas

Domino 5.*x* provides a powerful application development environment in large part because a number of programming languages can be used exclusively, or in conjunction with one another, to build powerful and sophisticated applications. At the present time, Domino 5.*x* supports the following programming languages: Notes Formulas, LotusScript, Java, and JavaScript. Of the four, the Notes Formula language has been around since the earliest days of Notes and provides a fairly simple interface for programming Domino applications. In fact, some coding tasks in Notes can be accomplished only using the Formula language!

According to the Lotus Designer Help, a formula is "an expression that has program-like attributes." In Domino 5.*x*, some of the many things you can do with formulas are

- Create selection criteria for a view.
- Return a value to a field.
- Validate a field.
- Manipulate the value of a field.
- Create new fields in a document.
- Perform actions when documents are open, refreshed, or closed.
- Create replication formulas.
- Return a value in a view column.
- Automate buttons or hotspots and code agents.

USE CONSTANTS IN FIELDS AND FORMULAS

In Domino, formulas are used to

- Automate tasks.
- Act on a condition.
- Compare values.
- Compute values.
- Modify values.
- Create selection criteria: views, agents, and replication formulas.

A formula is composed of one or more statements, which consist of any of the elements listed in Table 4.1.

TABLE 4.1

FORMULA ELEMENTS

Element	*Description*
@Commands	Similar to functions, @Commands execute Notes commands, most of which duplicate menu options such as File, Save.
@Functions	Prebuilt functions that perform a specific action and return a result.
Constants	Static values that do not change. Notes supports three types of constants: Text, Number, and Date. The following is an example of a text constant: "Samuel Hatter."
Keywords	Statements that perform special functions. There are five: `DEFAULT`, `ENVIRONMENT`, `FIELD`, `REM`, and `SELECT`.
Operators	Operators assign values and modify values. Domino supports a large number of operators.

The remainder of this section is not intended to be a comprehensive compendium of formula functions and commands; there are far too many! Rather, it is intended to provide an overview of how they are used. You can refer to the Domino Designer Help database `Help5_Designer.nsf`.

Computing Values Using Formulas

One of the main uses for the Formula language is to compute values for an application. When computing values, there a four ways to get data used for the computations: Constants, Fields, new values created with operators, and temporary variables. Each is explained in the following sections.

Using Operators to Compute New Values

As mentioned before, operators can be used to combine and compare values. Table 4.2 lists all the Formula language operators and their precedence, meaning the order in which they are operated upon, or, in other words, which operations are performed first.

TABLE 4.2

FORMULA OPERATORS

Operator	*Operation Performed*	*Precedence*
:=	Assignment	NA
:	List concatenation	1
+	Positive	2
-	Negative	
*	Multiplication	3
**	Permuted multiplication	
/	Division	
*/	Permuted division	
+	Addition, concatenation	4
*+	Permuted addition	
-	Subtraction	
*-	Permuted subtraction	
=	Equal	5
*=	Permuted equal	
<>	Not equal	
!=	Not equal	
=!	Not equal	
><	Not equal	
*<>	Permuted not equal	
<	Less than	
*<	Permuted less than	
>	Greater than	
*>	Permuted greater than	
<=	Less than or equal	
*<=	Permuted less than or equal	
>=	Greater than or equal	
*>=	Permuted greater than or equal	
!	Logical NOT	6
&	Logical AND	
|	Logical OR	

The following examples illustrate the use of some of the more common operators.

Assignment:

```
tTemp:="Ted Nugent";
```

Concatenating text:

```
"Welcome to our site" + tWebUser+ "!";
```

Creating a text list:

```
"Abraham Lincoln":"Ronald Reagan":"George Bush";
```

Adding numbers:

```
nSubtotal+nTax;
```

Subtracting numbers:

```
nSubtotal-nDiscount;
```

Dividing numbers:

```
nTotal/nTax;
```

Multiplying numbers:

```
nSubtotal*nRate
```

Equality:

```
@if(@UserName="Dave Hatter/Definiti";@Prompt([OK];"Authorized";"User
➥Authorized");@Return(@Prompt([OK];"Error";"Not Authorized")));
```

Inequality:

```
@if(@UserName<>"Dave Hatter/Definiti";@Return(@Prompt([OK];
➥"Error";"Not Authorized")); @Prompt([OK];"Authorized";"User
➥Authorized"));
```

Greater than:

```
@if(nSubTotal>100000;@Failure("Order entry error");@Success);
```

Greater than or equal to:

```
@if(nSubTotal>=100000;@Failure("Order entry error");@Success);
```

Less than:

```
@if(nSubTotal<1;@Failure("Order entry error");@Success);
```

Less than or equal to:

```
@if(nSubTotal<=1;@Failure("Order entry error");@Success);
```

Logical AND:

```
@if(nSubTotal<1 & nTotal="";@Failure("Order entry error");@Success);
```

Logical OR:

```
@if(nSubTotal<1 ¦ nTotal="";@Failure("Order entry error");@Success);
```

Logical NOT:

```
@if(! nSubTotal<1;@Success;@Failure("Order entry error"));
```

Using Constants in Formulas

As mentioned earlier, constants are static values that do not change. In Notes and Domino, they come in three flavors: Text, Number, and Date. Each is illustrated as follows:

- **Text.** Text constants should be enclosed in double quotes—for example, `"Wyatt Hatter"`. If the string contains embedded double quotes, the backslash character should be used as an escape character—for example, `"Samuel said \" Merry Christmas\"!"`. A text list can also be represented as a constant: `"Samuel Hatter":"Wyatt Hatter":"Leslee Hatter"`.
- **Number.** Number constants are represented simply as numbers—for example, 8, -23, 100.002, 18E2. Be sure *not* to use quotes or the numbers will be interpreted as text constants. Number lists can be represented as a constant as well: 1:2:3:4:5.
- **Date-Time.** Date-time constants are used to represent date /time values and should be enclosed in square brackets ([].) You can use “AM” and “PM” in the constant to indicate time in a 12-hour format—for example, [18:23], [6:45PM], [12/21/1995], [12/05/1998 12:39PM]. Like text and number lists, date-time lists can also be represented by constants: [01/01/1900]:[01/01/2000].

The following example illustrates using a text constant in an action formula to determine whether the current user is authorized to perform the action.

```
@if(@IsMember(@Name([CN];@UserName);"Samuel Hatter":"Wyatt Hatter");
➥"";@Prompt([OK];"Authorization Error";"You are not authorized to
➥perform that operation."));
```

Using Temporary Variables

Temporary variables are used to temporarily store values that exist only within the currently executing formula. (When the code ends, the variable is no longer available.) Variables can go a long way toward making your formulas more readable and more modular. A temporary variable need not be declared before it is used, and it is instantiated simply by using the assignment operator `":="`. The following example illustrates using a temporary variable to hold a list of values returned from a lookup and then testing the variable for errors before using its value.

```
List:=@Dblookup("";"Gonzo/Definiti":"names.nsf";"People";"Samuel
➥Hatter";2);
@If(@IsError(List);"Error in lookup";List);
```

Using Fields in Formulas

Formulas have access to fields in the current document, which may be the document that is currently open in a form, a document selected in a view, or selected documents that an agent runs against. To use a field in a formula, you simply reference the name of the field in the formula. For example, if you want to compute the value of the `nTotal` field based on the values in the `nSubTotal` and the `nTax` fields, you might make the `nTotal` field computed and enter the following formula:

```
nSubTotal+nTax
```

When the form containing the `nTotal` field is opened, refreshed, or saved, the value is recalculated.

The next example illustrates sending an email using the Formula language to users in a field named `tRecipients`. The contents of the `tSubject` field provide the email's subject.

```
@MailSend( tRecipients ; "";"" ; tSubject );
```

USING FORMULAS IN FORMS

Forms provide the structure for documents. They're used to create, modify, and display the data contained in documents. Because forms are the primary tool used to create and modify data, they are highly programmable, and formulas play a large role. This section focuses on writing formulas in forms. You can use formulas to do many things in a form, including the following:

- Set the window title for a form.
- Validate editable fields before a document is saved.
- Set the default value of an editable field.
- Calculate the value of a computed field.
- Provide the value for a keyword field.
- Send a mail message based on certain conditions.
- Set HTML attributes of a form or field.
- Perform tasks when document is opened, saved, or recalculated.
- Set the target frame of a form.
- Provide help.
- Set the value of computed text.

The places where the Formula language can be used in a form are defined in the following list:

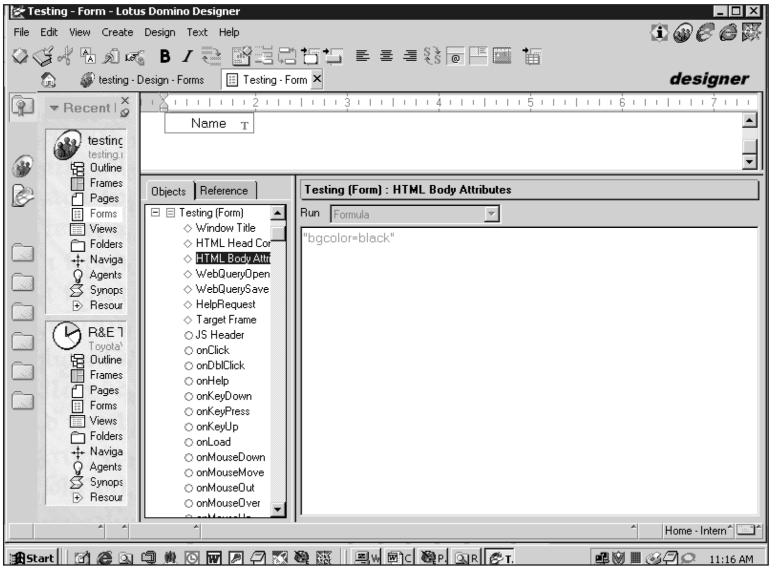
- Window Title
- HTML Head Content
- HTML Body Attributes
- Web Query Open
- Web Query Save
- Help Request
- Target Frame
- `Queryopen`
- `Postopen`
- `Querymodechange`
- `Postmodechange`
- `Postrecalc`
- `Querysave`
- `Postsave`
- `Queryclose`

- Default field values
- Field input translation
- Field validation
- Field HTML attributes

Creating a Formula in a Form

Like any design element, a form is an object that contains other objects, most of which are programmable in some way. The following list provides a generic overview of the steps involved in adding a formula to a form.

1. Open the form.
2. Select the element to which you want to add code from the Objects tab shown in Figure 4.1.
3. Enter the formula in the Programmer's pane.
4. Save the form and test it.



"bgcolor=black"

FIGURE 4.1
Adding a formula to a form's HTML Body Attributes element.

Because there are so many ways to use formulas in forms, this section covers only the most common usage.

Computing the Window Title

By default, the title of a form (or page for that matter) is "Untitled," which is not very descriptive or useful. You can use the Formula language to set the form title, which is a very good habit to get into. Just follow these steps:

1. Open the form.
2. Select the Window Title element on the Objects tab.
3. Enter the formula in the Programmer's pane.
4. Save the form.
5. Test the form.

The following example illustrates the creation of a conditional window title:

```
@If(@isNewDoc;"New Contact";"Contact:" +tLastName +
➥@if(!@Trim(tFirstname)="";", "+tFirstName;""))
```

Using Computed Values in a Form

Computed values can be used to make forms more dynamic and interactive without requiring the database to store unnecessary information. Two types of computed values are not stored (contrast with `Computed` and `Computed When Composed` fields): computed text and computed for display fields.

Computed text can be used whenever you want to dynamically display information on the screen, but don't necessarily need to use the value elsewhere, such as in a computed or editable field. Computed text can be formatted with any of the normal text formatting options. The only real restriction on computed text is that the formula used to compute it must return a text value. To create computed text, follow these steps:

1. Open the form in the Designer.
2. Position the cursor where the text should appear.
3. Select Create, Computed Text from the menu.
4. Enter your formula in the Programmer's pane. Remember that it must evaluate to text.
5. Optionally format the text.
6. Save the form.
7. Test the form.

The following formula is an example of a computed text formula:

```
"Welcome visitor" + @Environment("Counter")
```

If you need to use a computed value in other formulas, or if the value is not text and does not lend itself to text conversion, you can use a `Computed for Display` field. `Computed for Display` fields have all the features and characteristics of the other types of computed fields, but their values are not stored in the document.

When using computed fields, keep the following in mind:

- `Computed for Display` fields (or any computed field) must have a formula.
- `Computed for Display` fields re-execute their formulas when the form is opened, refreshed, or saved.
- The values produced by `Computed for Display` fields are not stored.
- `Computed for Display` fields can be used to provide values to other formulas.

To create a `Computed for Display` field, follow these steps:

1. Open the form in the Designer.
2. Position the cursor where the field should appear.
3. Select Create, Field from the menu.
4. Name the field.

5. Set the field properties; be sure to set the proper data type for the value your formula is to return.
6. Enter your formula in the Programmer's pane.
7. Optionally format the text.
8. Save the form.
9. Test the form.

The following example illustrates using a formula to compute the value of a `Computed for Display` field that displays a total. Because the total is easily calculated and should be recalculated whenever the other fields change, this would be a good use for a `Computed for Display` field.

```
nSubtotal+nTax+nShipping
```

Computing the Values for a Keyword Field

One of the most useful ways to apply formulas in a form is to provide the values for a keyword list field. There are two basic ways to do this: using `@DbColumn` or `@DbLookup`. `@DbColumn` returns a column of values from a view, whereas `@DbLookup` returns a column of values (or a field) for all documents in a view that match a supplied key. Using one of these two formulas enables you to dynamically populate keyword fields.

Using *@DbColumn*

If you want to populate a list with values from a single column in a view, you should use `@DbColumn`.

```
@DbColumn( class : mode ; server : database ; view ; columnNumber )
```

The first parameter that `@DbColumn` takes is `class:mode`. This parameter is a text list. The first element in the list, `class`, specifies the type of database being accessed (Notes or ODBC). If the database is a Notes database, you can pass an empty string (`""`). The second element in the list, `mode`, determines whether the results should be cached or not. Indicate *`NoCache`* to force the results to be looked up each time the function is called. Although this method is slower, it ensures the most current list.

The second parameter, `server:database` is also a text list. It specifies the server and the database that contains the view that should be used to provide the result set. If the view is in the current database, you can pass an empty string (`""`). You can also use the replica ID of a database rather than specifying the name of the server and database—for example, 8525682F:0070B14.

The third parameter, `view`, specifies the name of the view to use for the lookup.

TIP

Although you can supply either the view name or alias for this parameter, it is generally preferred to use the view alias so that if the view name changes, the formulas still work.

The final parameter, `columnNumber`, is a number value that represents the column that contains the values to return. When determining which column number to use, be sure to count hidden columns. For example, if the column is the second column displayed on the screen, but there is a hidden column between the first and second column, the correct value for this parameter would be 3.

TIP

If you provide a column number in a lookup formula, you should not count hidden columns, columns that use a constant for their formulas (such as “1”), or columns that use any of the following formulas: `@DocChildren`, `@DocDescendants`, `@DocLevel`, `@DocNumber`, `@DocParentNumber`, `@DocSiblings`, `@IsCategory`, and `@IsExpandable`.

The following example illustrates using `@DbColumn` to retrieve a list of user names from a view in the Domino Directory named People:

```
View:="People";
Col=1;
List:=@DbColumn("":"NoCache";"Gonzo/Definiti":"names.nsf";View;Col);
@If(@IsError(List);"Lookup Error";List)
```

Using @DbLookup

If you want to populate a list with values from a column in a view where the documents match a certain key, rather than all documents in the view, you should use `@DbLookup`. Keep in mind that for the lookup to work, the first column in the view must be sorted.

```
@DbLookup( class : mode ; server : database ; view ; key;
➥columnNumber )
```

```
@DbLookup( class : mode ; server : database ; view ; key; fieldName )
```

The first three parameters of `@DbLookup` are identical to the earlier `@DbColumn`.

The fourth parameter, `key`, is a text value that specifies how the returned values should be filtered.

The final parameter can either be a number representing the column to return (`columnNumber`) or a text value (`fieldName`) representing a field in the document. Only documents that match the value specified in the `key` parameter will have their values returned.

The following example illustrates using `@DbLookup` to search a view sorted on department, which returns a list of people whose department is IT:

```
View:="People";
Key:="IT";
Col=2;
List:=@DbLookup("";"";View;Key;Col);
@If(@IsError(List);"Lookup Error";List)
```

Follow these steps to add a formula to a keyword field:

1. Open the form in the Designer.
2. Position the cursor where the keyword field should appear.
3. Select Create, Field from the menu.
4. Name the field.
5. Set the field type to Dialog List, Radio Button, Check Box, List Box, or Combo Box.
6. Click the Control tab and select Use Formula for Choices.
7. Save the form.
8. Test the form.

Computing Stored Values with Formulas

Formulas can also be used to compute values that are to be stored in a document. To store values in a document, you must create one of the following field types:

- Computed
- Computed When Composed
- Editable

Using Formulas in Computed Fields

Computed fields are an important resource for developers. Not only are their values stored when a document is saved, but they can be used in other formulas and fields.

`Computed When Composed` fields are computed only once, when the form is opened for the first time. They are useful when you want to store a value, but its value is recomputed. For example, the original author of a document or the time and date the document was originally created.

Computed fields are the most flexible and useful of the three types of computed fields because they are stored and recomputed whenever a form is opened, refreshed, or saved. They are best suited for storing the results of computations involving other fields.

To create a computed field, follow these steps:

1. Open the form in the Designer.
2. Position the cursor where the computed field should appear.
3. Select Create, Field from the menu.
4. Name the field.
5. Set the type to Computed.
6. Enter a formula for the field in the Programmer's pane.
7. Save the form.
8. Test the form.

Using Formulas in Editable Fields

Editable fields are the primary tools for gathering, storing, and modifying information from users. Editable fields can be enhanced by the use of formulas in numerous ways. Formulas can be used to

- Set a default value for an editable field when a document is created.
- Reformat or mask data in a field.
- Validate the data a user has entered in a field before the document is saved.

Setting a Default Value

You can use the Formula language to specify a default value for an editable field. To supply a default value, follow these steps:

1. Open the form in the Designer.
2. Select or create the field for which you want to add a default value.
3. Choose Default Value on the Objects tab.
4. Enter a formula in the Programmer's pane.
5. Save the form.
6. Test the form.

Using Translation Formulas to Format Field Values

If you want to ensure that data in certain fields is formatted properly, you can use an input translation formula. When a document is saved or refreshed, the input translations formulas are executed. Input translation formulas can do a number of tasks such as

- Trim leading and trailing spaces off a value (`@Trim`).
- Make a value all uppercase or lowercase (`@UpperCase` or `@LowerCase`).
- Make a value, such as a name, a proper noun (`@Propercase`).
- Provide a format to values such as zip codes or phone numbers by inserting characters such as hyphens, parentheses, periods, and/or commas.

To add an input translation formula to a field, follow these steps:

1. Open the form in the Designer.
2. Select or create the field for which you want to add an input translation formula.
3. Select Input Translation on the Objects tab.
4. Enter a formula in the Programmer's pane.
5. Save the form.
6. Test the form.

The following example demonstrates a simple input translation formula that trims leading and trailing spaces off the value in a field:

```
@Trim(tLastName)
```

Using Input Validation Formulas

Input validation formulas provide a means to ensure that users enter correct and complete data by stopping users from saving a document if a field's value is incorrect. Every time a document is saved or refreshed, the input validation formulas are executed to ensure that the proper values have been entered. For each field, the value is tested according to the formula, and if the value is acceptable, the formula returns true and the document can be saved. If a value is unacceptable, the formula returns false and the document cannot be saved until the proper data is entered. Additionally, the user is prompted with a message supplied by the developer to explain the problem.

When creating input validation formulas, you must use the `@Failure` and `@Success` functions. `@Success` returns true and allows the document to be saved and to continue. `@Failure` returns false and displays a prompt with a message the developer supplies as a parameter. Additionally, it causes the termination of the save.

To create an input validation formula, follow these steps:

1. Open the form in the Designer.
2. Select or create the field for which you want to add an input validation formula.
3. Select Input Validation on the Objects tab.

4. Enter a formula in the Programmer's pane.
5. Save the form.
6. Test the form.

The following example demonstrates an input validation formula that tests the `tLastname` field to ensure that it is not empty. If `tLastname` is empty, a prompt displays the message `Please enter a value for last name.` and the save is aborted.

```
@if(@Trim(tLastName)="";@Failure("Please enter a value for last
➥name.");@Success);
```

USING FORMULAS IN VIEWS

As you know from Chapter 2, views present a sorted and/or categorized list of documents, and serve as the primary way to access data in a Notes database. The Formula language is the primary development language used to build views and are used to create view selection criteria, to add logic to view columns, and to perform actions based upon user-generated events such as opening and closing a view or pasting a document into a view. This chapter focuses on writing formulas in views. The places where the Formula language can be used in a view are defined in the following list:

- View Selection
- View Columns
- Form Formula
- Help Request
- Target Frame (single click)
- Target Frame (double click)
- `Queryopen`
- `Postopen`
- `Regiondoubleclick`
- `Queryopendocument`

- `Queryrecalc`
- `Queryaddtofolder`
- `Querypaste`
- `Postpaste`
- `Querydragdrop`
- `Postdragdrop`
- `Queryclose`

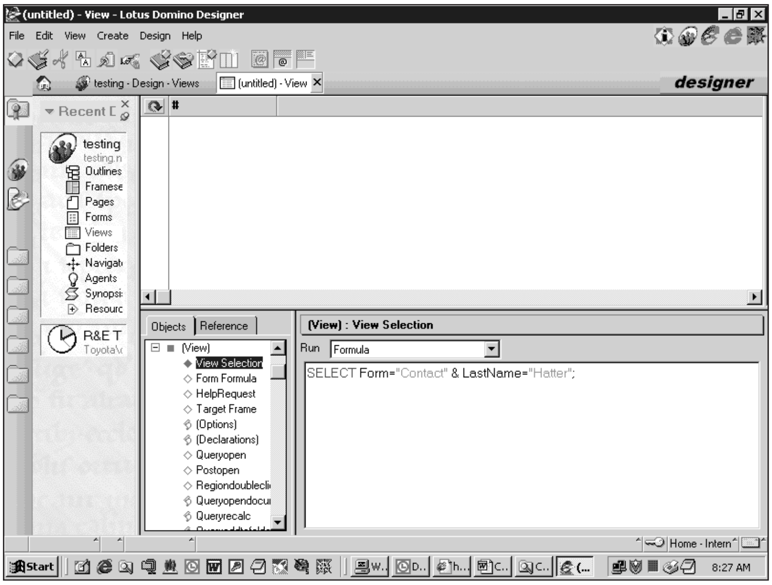
Creating View Selection Formulas

By default, a view displays all documents in the current database. In most cases, this is not what a developer wants to do, hence, the view selection formula. View selection formulas enable you to select a subset of documents by specifying criteria that a document must meet before it is included in a view. After a view selection formula has been applied to a view, all documents are tested against the formula. If a document meets the criteria, the selection formula returns true (1) and the document is included in the view. Otherwise, false (0) is returned and the document is excluded from the view.

A view selection formula has two basic parts: the `SELECT` keyword, which indicates that the view should be filtered, and the formula that tests the desired criteria and returns true or false for each document.

To create a view selection formula, follow these steps:

1. Open the View window in Domino Designer (see Figure 4.2).
2. Select View Selection from the Objects tab in the Programmer's pane.
3. Select Formula in the Run drop-down list on the code window.
4. Code your selection formula.
5. Save and test the view to ensure that the proper documents are selected.



```
SELECT Form="Contact" & LastName="Hatter";
```

FIGURE 4.2
The View design window in Designer 5.x.

Table 4.3 illustrates the use of view selection formulas.

TABLE 4.3

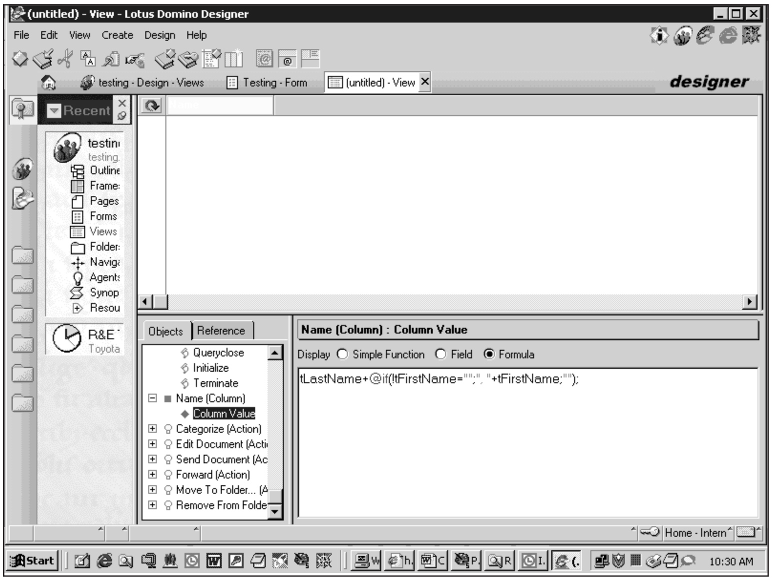
VIEW SELECTION FORMULA EXAMPLES

Selection Formula	*Selects Documents*
`SELECT @All`	Selects all documents in the database. This is the default selection formula.
`SELECT @Contains(@ProperCase(tLastName);"Hatter");`	Selects all documents where the field `tLastname` contains “Hatter”.
`SELECT nTaxRate>=.33`	Selects all documents where the `nTaxRate` field contains values greater than or equal to 33%.

Selection Formula	*Selects Documents*
`SELECT @Created <= [01/01/2000]`	Selects all documents that were created on or before 01/01/2000.
`SELECT nTaxRate>=.33 & nOverTaxed=1`	Selects all documents where the field `nTaxRate` is greater than or equal to 33% and the field `nOvertaxed` is true.

Creating View Column Formulas

View columns provide the mechanism to display values from the underlying documents in a view. A column can either contain a value from a field in a document, or it can contain a computed value that may or may not be based on fields. The most simple column formula is one in which a single field value is displayed. You can either select the field from the Database Fields list on the Reference tab, or you can type it in directly. For example, to display the contents of the `tLastName` field, simply enter **tLastName**. Figure 4.3 illustrates column design.



Name (Column) : Column Value

Display ( ) Simple Function ( ) Field [✓ Formula]

```
tLastName+@if(!tFirstName="";", "+tFirstName;"");
```

FIGURE 4.3
A view column in design mode.

Although using fields in views is a common and simple practice, the Formula language enables you to build very complex (and useful) formulas for columns. The first thing that you must remember when building column values is that values displayed in a column must be of the same data type, or must be converted to the same type. For example, if you wanted to display someone's name (text) and birthday (date-time) in the same column, you must convert the date to text value and then concatenate the values in the column. Table 4.4 lists the most common and useful @Functions for converting values from one data type to another. For more detailed information on these and other @Functions, see the Domino Designer Help database (`help5_designer.nsf`).

TABLE 4.4

FUNCTIONS FOR TESTING AND MANIPULATING DATA TYPES

Function	*Action*
`@Text(value)`	Converts the supplied value to a text string.
`@Text(value; format)`	Converts the supplied value to a text string according to a specified format.
`@TextToNumber(value)`	Converts the supplied text string to a number.
`@TextToTime(value)`	Converts the supplied text string to a date-time.
`@IsText(value)`	Returns true (1) if the supplied value is a text string or text string list.
`@IsNumber(value)`	Returns true (1) if the supplied value is a number or number list.
`@IsTime(value)`	Returns true (1) if the supplied value is a time-date or time-date list.

Table 4.5 contains examples illustrating column formulas.

TABLE 4.5

EXAMPLE COLUMN FORMULAS

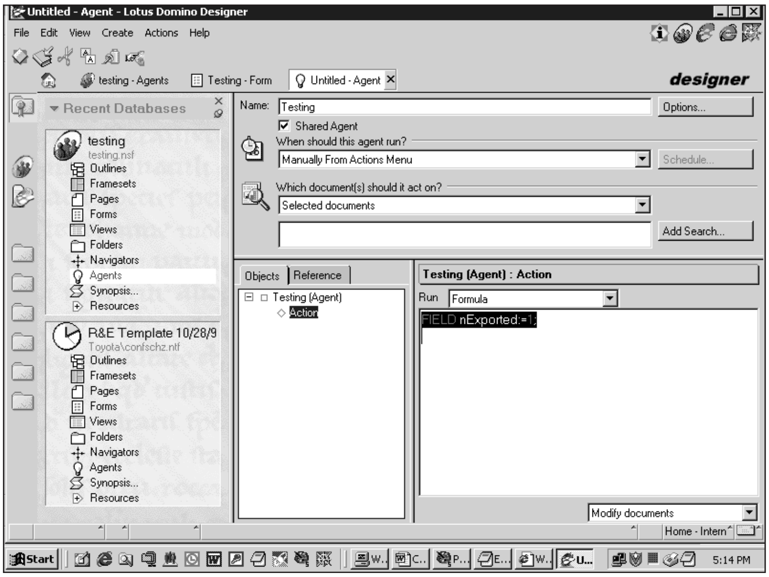
Formula	*Result*
`tLastName + ", " + tFirstName`	Displays the contents of the `tLastname` field followed by a space and a comma and the contents of the `tFirstName` field—for example: `Hatter, Samuel`
`"Opened: " + @Text(@Created;"S0D0")`	Displays the text constant `"Opened: "` and the date-only portion of the date and time the document was created—for example: `"Opened: 11/27/99"`
`"Assessed Value = "+ @Text(nAValue;"C,2")`	Displays the text constant `"Assessed Value = "` and the value of the `nAValue` with two decimal places and a currency symbol—for example: `"Assessed value = $123,000.00"`
`"Total: " + @Text(nAValue * nRate;"C,2")`	Displays the text constant `"Total: "` and the product of `nAValue * nRate` with two decimal places and a currency symbol—for example: `"Total: $888.23"`
`@TexttoTime(@Right(tCreatedOn;8))`	Reads the right 8 characters of the `tCreatedOn` field and converts them to a date, which is displayed in the column.
`@TexttoNumber(tPublish)`	Converts the `tPublish` field to a number and displays it in the column.

USING FORMULAS IN AGENTS

As you know from Chapter 3, agents are self-contained blocks of code that can perform a number of tasks throughout an application. In many cases, the Formula language in an agent is the best tool to accomplish simple tasks quickly and easily. For example, creating an agent that processes selected documents and updates a field named `nExported` takes only one line of code with the Formula language! Figure 4.4 shows an agent in design mode.

To use the Formula language in an agent, follow these steps:

1. Open the database in the Domino Designer.
2. Select Agents in the Navigator pane.
3. Click New to create a new agent or double-click an existing agent.
4. Select Formula in the Run box in the Programmer's pane.
5. Enter a formula in the Programmer's pane.
6. Save the agent.
7. Test the agent.



Name: Testing

☑ Shared Agent

When should this agent run? Manually From Actions Menu

Which document(s) should it act on? Selected documents

Run: Formula

```
FIELD nExported:=1;
```

Modify documents

FIGURE 4.4
Building an agent in the Designer.

The following code example illustrates a simple agent that sets a field in selected documents based on the value of another field:

```
@if(nExported=1;@SetField("nArchive";"1");@SetField("nArchive";0));
```

Using Formulas in Actions

As you know from Chapter 3, actions are handy tools that can be used to automate user tasks and can be used in forms and views. And Domino 5.*x* adds shared Actions, enabling you to write code once and share it across multiple Views and Forms—very cool!

To use the Formula language in actions, follow these steps:

1. Open the database in the Domino Designer.
2. Open the form or view that contains the action, or open the shared action from the database resources.
3. Click New to create a new action, or double-click an existing action.
4. Select Formula in the Run box in the Programmer's pane.
5. Enter a formula in the Programmer's pane.
6. Save the action.
7. Test the action.

The following example demonstrates using the Formula language in an action to print the current document or view:

```
@Command([FilePrint])
```

Understanding New and Enhanced @Functions

In the past, the venerable Formula language has been expanded and enhanced in each new release of Notes, and R5 is no exception. Table 4.6 displays the new and enhanced @Functions in R5.

TABLE 4.6

NEW AND ENHANCED @FUNCTIONS

Function	*Type*
`@BrowserInfo`	New
`@ValidateInternetAddress`	New
`@UserNamesList`	New
`@AddtoFolder`	New
`@UndeleteDocument`	New
`@SetTargetFrame`	New
`@Name`	Enhanced
`@PickList`	Enhanced

New @Functions

The `@AddtoFolder`, `@BrowserInfo`, `@SetTargetFrame`, `@UndeleteDocument`, `@UserNamesList`, and `@ValidateInternetAddress` are new @Functions. Each function is explained in the following sections.

@AddtoFolder

This function can be used in SmartIcons, button formulas, and agent formulas, and it can perform one of three possible actions:

- Add current document to a folder.
- Add current document to one folder while removing it from another.
- Remove current document from one folder.

Which action is performed depends on which parameters are supplied when the function is called. The following example shows the functions syntax:

```
@AddToFolder(foldernameadd ; foldernameremove);
```

The `FolderNameAdd` parameter is a string that indicates the folder into which the document should be added. The `FolderNameRemove` parameter indicates the name of the folder from which the document should be removed. To perform only one of the actions, an empty string (`""`) can be substituted for either argument. The following example illustrates using this function to add the current document to a folder without removing it from any other folders:

```
@AddToFolder("Contacts" ;"");
```

@BrowserInfo

The `@BrowserInfo` function returns information about the current client (Notes or Web browser). `@BrowserInfo` can be used in any formula with the exception of view selection column formulas. When opening forms using this formula, non-R5 Notes clients receive the error message `Invalid formula: unknown function/operator.` . Using the `@Version` formula to test the client version before executing this formula can prevent this situation. The following example illustrates how the `@BrowserInfo` function is used:

```
@BrowserInfo("propertyname")
```

The function takes one parameter, `propertyname`, which indicates which browser property you'd like to view. Table 4.7 lists and explains each of parameters supported by this function.

TABLE 4.7

@*BROWSERINFO* PROPERTIES

Property	*Type*	*Return Value for Browsers*	*Return Value for Notes Client*
BrowserType	Text	Browser type: "Compatible", "Microsoft", "Netscape", "Unknown".	Notes
Cookies	Boolean	1 is cookie support, else 0.	0
DHTML	Boolean	1 if DHTML support, else 0.	0

continues

TABLE 4.7 continued

Property	*Type*	*Return Value for Browsers*	*Return Value for Notes Client*
FileUpload	Boolean	1 file uploads are supported, else 0.	0
Frames	Boolean	1 if `<FRAME>` tag is supported, else 0.	1
Java	Boolean	1 if Java applet support, else 0.	1
JavaScript	Boolean	1 if JavaScript support, else 0.	1
Iframe	Boolean	1 `<IFRAME>` tag is supported, else 0.	0
Platform	Text	The operating system the browser is running on: “Win95”, “Win98”, “WinNT”, “MacOS”, or “Unknown”.	Unknown
Robot	Boolean	1 if browser is a Web robot, else 0.	0
SSL	Boolean	1 if SSL support, else 0.	0
Tables	Boolean	1 if `<TABLE>` tag is supported, else 0.	1
VBScript	Boolean	1 if VBScript support, else 0.	0
Version	Number	The browser version number. -1 for unrecognized browsers.	Notes client build number

In the following example, calling the `@BrowserInfo` function with the `"Tables"` property returns 1 in most current browsers and the Notes client:

```
@BrowserInfo("Tables");
```

@SetTargetFrame

The `@SetTargetFrame` function enables you to explicitly specify a target frame when opening a form, page frameset, or view, or when creating or editing a document. The syntax of the `@SetTargetFrame` function is

```
@SetTargetFrame(targetframe);
```

The `@SetTargetFrame` function takes one parameter, `"TargetFrame"`, which indicates the frame into which the element should be opened. This parameter works in one of two ways. Supplying a text value that is a valid frame name indicates the frame into which the element should be opened. A text-list can be supplied if the element should be opened in a frameset containing nested frames. This function can be used in action and hotspot formulas. An example of the `@SetTargetFrame` function follows:

```
@SetTargetFrame("Body");
@Command([Openpage];"Welcome");
```

@UndeleteDocument

The `@UndeleteDocument` function can be used to remove the soft delete status of a document in databases that have the `"Soft Delete"` property enabled. It can be used in SmartIcons, button, or agent formulas in a view that is designed to display "soft deleted" documents. The `@Undelete` function syntax is shown as follows:

```
@UndeleteDocument
```

@UserNamesList

This handy new function returns a text list containing the current username, any group names, and any roles under which the user has been granted access. The following example shows the function's syntax:

```
@UserNamesList
```

Keep in mind that this function works only for a database on a server or a local database with "Enforce a Consistent Access Control List Across All Replicas" in effect. It does not work in column, selection, mail agent, or scheduled agent formulas. On local databases that do not enforce consistent ACLs, this function returns an empty string (`""`). The following example illustrates using this function:

```
UserList:=@UserNamesList;
@If(@IsMember(@Name([CN];@UserName);UserList);@Prompt([OK];
➥"Membership";"User is a member.");@Prompt([OK];"Membership";
➥"User is not a member."));
```

@ValidateInternetAddress

This function can be used to evaluate a string to determine whether it is a valid Internet mail address. If the function returns NULL, the address was successfully validated; otherwise an error message indicating why the address could not be validated is returned.

```
@ValidateInternetAddress([Keyword]; Address)
```

`@ValidateInternetAddress` validates mail addresses based on the RFC 821 and RFC 822 standards, each of which is shown as follows:

- RFC 821: `dhatter@davehatter.com`
- RFC 822: "Last, First" `<Flast@davehatter.com>`

The following illustrates using the `@ValidateInternetAddress` function in a field. Validate formulas to ensure that the user enters an email address in the either the RFC 821 or RFC 822 format.

```
RFC821:=@ValidateInternetAddress([RFC821];mailaddress);
RFC822:=@ValidateInternetAddress([RFC822];mailaddress);
Failure:="Please enter a valid e-mail address.";
@if(RFC821=NULL;@If(RFC822=NULL;@Success;@Failure(Failure));
➥@Failure(Failure));
```

Enhanced @Functions

The `@Name` and `@Picklist` functions now support new options in R5. Each function is explained in the following sections.

@Name

The `@Name` function has been significantly enhanced in R5. In addition to its old functionality, it can now perform the following functions as well:

- Return the local part of an Internet address (`[LP]` option).
- Return an Internet address in RFC 821 format (`[Address821]` option).
- Return the phrase part of an RFC 822 formatted Internet address (`[PHRASE]` option).
- Remove the common name part of a distinguished name and return the remaining parts (`[HeirarchyOnly]` option).

The following example shows the syntax of the `@Name` function:

```
@Name([action]; name);
```

The following example shows the use of the `@Name` function to return the phrase portion of an Internet mail address in RFC 822 format:

```
@Name([PHRASE];mailaddress);
```

@Picklist

`@Picklist` displays a modal dialog box containing either a specified view, from which the user can select one or more documents and which returns the specified column value from the selected documents, or it launches the Address dialog box, displaying documents from all available Domino Directories. When the user makes a selection, `@PickList` returns those names. Its various syntaxes are shown as follows:

```
@PickList( [Custom] : [Single] ; server : file ; view ; title ;
➥prompt ; column ; categoryname  )

@PickList( [Name] : [Single] )

@PickList([Room])

@PickList([Resource])

@PickList([Folders]; [Single]server:database)

@PickList([Folders]; [Shared] server:database)

@PickList([Folders]; [Private] server:database)

@PickList([Folders]; [NoDesktop]server:database)
```

R5 offers two enhancements to `@PickList`. First, in a custom picklist (the `[custom]` parameter is specified), you can use the ReplicaID of a database in place of the `server` and `database` parameters. Additionally, a new parameter, `categoryname`, has been added for custom picklists. The `category-name` parameter enables you to limit the documents displayed in the picklist to a particular category in a view.

The following example illustrates using a custom picklist to display only one column in the specified view:

```
List:=@PickList([custom];"123456C1:09876512"; "Lookup"; "Contacts";
➥"Select a contact"; 2; "Hot");
```

UNDERSTANDING NEW AND ENHANCED @COMMANDS

R5 includes two new or enhanced @Commands, shown in the Table 4.8. Each is explained in the subsequent sections.

TABLE 4.8

NEW AND ENHANCED @COMMANDS

Command	*Type*
`@Command([CalendarFormat])`	Enhanced
`@Command([RefreshParentNote])`	New

@Command([CalendarFormat])

In R5, the `@Command([CalendarFormat])` command has been changed to allow one of five calendar formats, based on the parameter supplied. The syntax of the command is shown in the following example. Table 4.9 then illustrates each of the five valid parameters and their effect on the calendar format.

```
@Command([CalendarFormat];number);
```

TABLE 4.9

***@Command([CalendarFormat])* PARAMETER VALUES**

Value	*Effect*
1	Displays one day in the calendar. This value is newly supported in R5.
2	Displays two days in the calendar.
7	Displays one week (7 days) in the calendar.
14	Displays two weeks (14 days) in the calendar.
30	Displays one month in the calendar.
365	Displays a year in the calendar. For Web based calendar displays only.

The following example opens the calendar with one day displaying:

```
@Command([CalendarFormat];1);
```

@Command([RefreshParentNote])

The `@Command([RefreshParentNote])` command updates the parent document with any values entered through a dialog box. This enables a designer to update the parent document and close the dialog box without pressing the OK button on the dialog box. Remember that this command works only in dialog boxes. The following example shows the syntax of this command:

```
@Command(RefreshParentNote]);
```

Troubleshooting Formulas

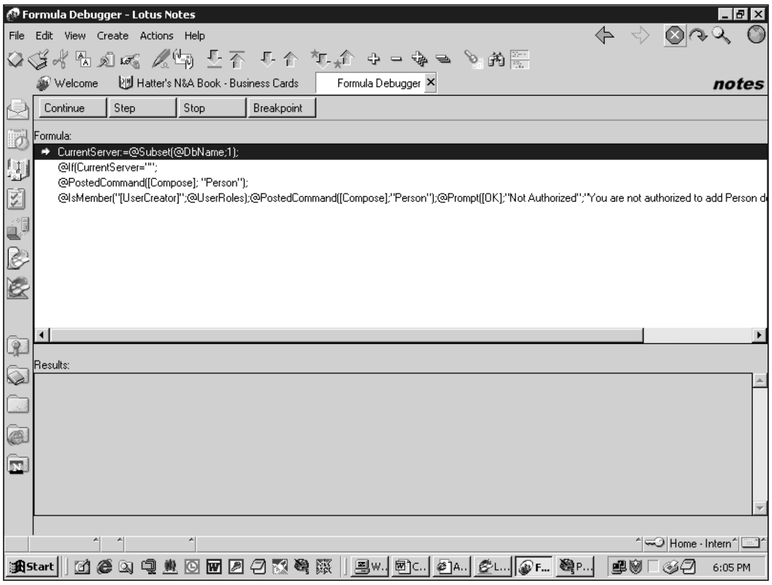
There are two basic ways to debug formulas. You can add statements to your formula (`@Prompt`) to follow its execution and display the value of variables, or you can use the very handy (but undocumented) formula debugger. Each is explained here.

The first method is fairly simple. Add `@Prompt` statements to stop the execution of the formula at certain points and to examine variables. The following example demonstrates this technique:

```
@Prompt([OK]; "Debug1"; "Before total calculation");
Total := nValue * nRate;
@Prompt([OK]; "Debug2"; "Total = " + @Text(Total))
```

Although this is very useful in most cases, for agents that run in the background (such as scheduled agents or mail paste agents) `@Prompt` does not work. Be sure to remove the debug statements after the code works as expected.

The second method relies on an undocumented (but very cool) feature that very few people know about: the “formula debugger.” To activate the formula debugger, hold down the Ctrl+Shift keys and select File, Tools, Debug LotusScript from the menu. The formula debugger is shown in Figure 4.5


Formula Debugger - Lotus Notes
File Edit View Create Actions Help
Welcome
Hatter's N&A Book - Business Cards
Formula Debugger
notes
Continue
Step
Stop
Breakpoint
Formula:
Results:
Home - Intern
Start
6:05 PM

```
CurrentServer:=@Subset(@DbName;1);
@If(CurrentServer="";
@PostedCommand([Compose]; "Person");
@IsMember("[UserCreator]";@UserRoles);@PostedCommand([Compose];"Person");@Prompt([OK];"Not Authorized";"You are not authorized to add Person d
```

FIGURE 4.5
The undocumented formula debugger.

After the formula debugger has been enabled, it works in much the same way as the LotusScript debugger. It enables you to step through the code line by line and view the values of variables and fields so that you can figure out what the code is doing. When you are finished using the forumla debugger, just repeat the same steps you used to enable it. Keep in mind that this is an undocumented and unsupported feature at this point and may not be as reliable and stable as you might hope.

What Is Important to Know

The following bullet points summarize the chapter and accentuate the key concepts to know for the exam.

- R5 has several new and enhanced @Functions and @Commands.
- Formula syntax is very important and can trip you up.
- The `@DbLookup` and `@DbColumn` formulas have not changed, but are very important and very useful.
- Know the difference between `Computed`, `Computer When Composed`, `Computed for Display`, and `Editable` fields.
- Framesets are a powerful new tool that make building Web apps with Domino much easier and more efficient.
- Know the difference between a view selection and a view column formula.
- Know how to add formulas to the various design elements that support them.