PROGRAMMER'S GUIDE

LOTUSSCRIPT RELEASE 3

Cross-Product BASIC Scripting Language

Contents

Index

Chapter 1
Introduction

LotusScript™ is a version of BASIC that offers not only the standard capabilities of structured programming languages like Pascal and C, but a powerful set of language extensions that enable object-oriented application development within and across products as well.

Learning About LotusScript

Lotus provides the following documentation for LotusScript:

- *The LotusScript Programmer's Guide*, a general introduction to LotusScript that describes the language's basic building blocks and how to put them together to create applications.
- *The LotusScript Language Reference*, a comprehensive summary of the LotusScript language, presented in A-Z format. *The LotusScript Language Reference* is available as online Help in all Lotus® products that support LotusScript and is also available in print.
- The documentation that accompanies each of the Lotus products that support LotusScript. This documentation describes the application development environment as well as the various extensions to the language that the individual product provides.

The *LotusScript Programmer's Guide*

The *LotusScript Programmer's Guide* covers the following topics:

- Chapter 1: Introduction

  A summary of the contents of this book and its typographic conventions.
- Chapter 2: Creating, Compiling, and Debugging Scripts

  The anatomy of a LotusScript application and the environment in which you create, run, debug, and save a LotusScript application.
- Chapter 3: Data Types, Constants, and Variables

  A survey of the kinds of values that LotusScript recognizes and the data structures you can use to manipulate those values in an application.

- Chapter 4: Procedures: Functions, Subs, and Properties

  How to write user-defined procedures to modularize the operations that an application performs.
- Chapter 5: Creating User-Defined Data Types and Classes

  How to create and use two kinds of data structure, the user-defined data type and the class.
- Chapter 6: Expressions and Operators

  How to build, evaluate, and perform operations on expressions.
- Chapter 7: Directing Traffic Within an Application

  How to manage flow control in an application through looping and branching operations.
- Chapter 8: Error Processing

  How to trap errors and make an application take appropriate action when it encounters an error.
- Chapter 9: Reaching Out

  How to make an application communicate with the end user, with other applications, and with the operating system.

Code examples in this book

This book contains numerous programming examples that illustrate specific features of LotusScript and show how you can use the language to perform a variety of common tasks. Each example consists of one or more lines of LotusScript code in the following order:

- Statements and directives that apply to a whole module (for example, a Use or Option Declare statement)
- Declarations and definitions of language elements intended to be available throughout a module (for example, a class definition or a module-level variable declaration)
- Executable code not contained in a procedure (for example, a Call statement)

The linear format in which code examples appear shows each example's underlying logic and design but only partly reflects the organization of a real-world LotusScript application. For instance, the following example prompts the user for an integer and then calls a procedure that displays the cube of that integer. This example contains an Option Declare statement, a module-level procedure definition, and three lines of executable code outside the procedure:

```
Option Declare
Sub PrintCube(X as Integer)
    Print X ^ 3
End Sub
Dim anInt As Integer
anInt% = CInt(InputBox$("Enter an integer: "))
Call PrintCube(anInt%)
```

Unlike the examples in this book, a LotusScript application is object-event driven: the application performs the operation you specify when the user opens a document, clicks a button, or enters a value in a text box (where document, button, and text box are objects, and opening, clicking, and entering are events). A LotusScript application typically consists of multiple event scripts, that is, different sets of LotusScript statements associated with different events for different objects. This makes for a modular, nonlinear organization of the application's code.

In striving to offer examples that are applicable to all products that support LotusScript, the *LotusScript Programmer's Guide* largely ignores the realities of object-event organization, because products may define different sets of objects and events and provide somewhat different programming environments.

This does not mean that you can't run the examples in the *LotusScript Programmer's Guide* in your Lotus product's programming environment. In general terms, you can turn an example in this book into a LotusScript application by creating an object and attaching the code to an event associated with that object. You run the example by triggering that event. The details of how you do this depend on the programming environment and the content and intent of the particular example.

If the product in which you are running LotusScript incorporates the LotusScript Integrated Development Environment (IDE), you could run the preceding example by doing the following:

1. Activate the Script Editor in the IDE for a new form.
2. Enter the Option Declare statement in Globals Options.
3. Enter the Dim statement in the Declarations section.
4. Enter the definition of PrintCube as a new sub.
5. Enter the last two statements of the example as the body of the Initialize sub.
6. Execute the example by loading the module (opening the document).

If the product in which you are are running LotusScript does not incorporate the IDE, a slightly different strategy is required. For example, in Lotus Forms™, the simplest way to run the preceding example is to do the following:

7. In the Designer, display the script editor and navigate to the Form object's Declarations procedure.
8. Enter the Option Declare statement, the definition of PrintCube, and the Dim statement in the Form object's Declarations procedure.
9. Enter the last two lines of the example in the Form object's NewForm procedure.
10. Switch to Filler mode to run the example.

See your product's documentation for details on writing and running LotusScript applications in its programming environment.

Typographical conventions

The *LotusScript Programmer's Guide* follows certain typographical conventions in its syntax diagrams and code examples. These conventions are summarized in the following two tables. The first table lists the conventions for syntax diagrams, and the second lists the conventions for code examples.

Typeface or character	*Meaning*	*Example and comment*
Bold	Items in bold must be entered as shown. Case is not significant.	**End** [*returnCode*] The keyword End is required.
Italics	Items in italics are placeholders for values that you supply.	**End** [*returnCode*] You can specify a return code by entering a value or expression after the keyword End.
Square brackets [ ]	Items enclosed in square brackets are optional.	**End** [*returnCode*] You can include a return code or not, as you prefer.
Vertical bar |	Items separated by vertical bars are alternatives: you can choose one or another.	**Resume** [**0** | **Exit** | *label*] You can include either 0, the keyword Exit, or a label, or none of these elements, as part of the Resume statement.

Continued

Typeface or character	*Meaning*	*Example and comment*
Braces { }	Items enclosed in braces are alternatives: you have to choose one. Items within braces are always separated by vertical bars.	**Exit** { **Do** | **For** | **ForAll** | **Function** | **Property** | **Sub** } You have to enter one of the following keywords after the keyword Exit: Do, For, ForAll, Function, Property, or Sub.
Ellipsis (...)	Items followed by an ellipsis can be repeated. If a comma precedes an ellipsis (,...), you have to separate repeated items by commas.	**ReDim** *arrayName*(*subscript*,...) You can specify multiple subscripts in a ReDim statement.

Item	*Convention*	*Example*
Apostrophe (')	Introduces a comment.	' This is a comment.
Underscore (_)	Signifies that the current line of code continues on the following line.	Dim anArray(1 To 3, 1 To 4) _ As String
Colon (:)	Separates discrete statements on the same line.	anInt% = anInt% * 2 : Print anInt%
Keyword	Begins with a capital letter. May contain mixed case.	Print UCase$("hello")
Variable	Begins with a lowercase letter. May contain mixed case.	anInt% = 5
Procedure	Begins with a capital letter. May contain mixed case.	Call PrintResults()

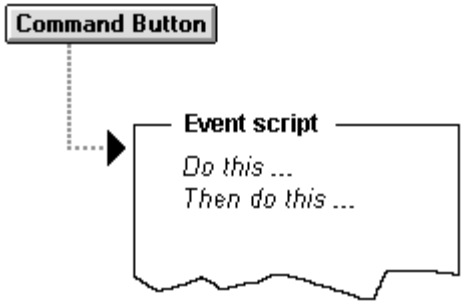
Chapter 2
Creating, Compiling, and Debugging Scripts

This chapter describes, in general terms, how to use your script editor to write and modify scripts, how to compile scripts, and how to use your debugger to locate problems in the logic of your applications. The environment in which you write, debug, and run scripts depends on your Lotus product. To learn about your product's programming environment, see your product documentation.

What Is a Script?

A **script** is composed of statements in the LotusScript language. An **application** is a collection of scripts that have been compiled and can be run by a user.

The Lotus product in which you are working provides objects that you use as building blocks to create an application. Each object has an associated set of events. Each **event** indicates that an action in an application has occurred. You write scripts to define responses to these events. For example, when the user clicks a command button, LotusScript runs the script that you defined for that command button's "click" event.



A Lotus product or the system can also initiate events. For example, a database sort operation is an event that is internal to a database application.

You write scripts in your script editor, using the LotusScript language and the properties, methods, and events defined for your product's objects. Some products can automate parts of the scripting process, restricting or eliminating the need to use parts of LotusScript. See your product documentation for more information.

Working in Your Script Editor

Use your script editor to view, write, and modify scripts. Your script editor includes standard editor features, such as cut, copy, and paste. You can also move from one script to another using the controls in your script editor.

You write a script in a space associated with an object and an event; LotusScript then attaches your script to the object and event. After you select the object and event to which you want to attach a script, type the instructions you want to execute when the event occurs. For more information on your product extensions, see the product documentation.

Note From the script editor in many Lotus products, you can highlight a product object's property or method and press **F1** to display a Help topic about that term. Similarly, you can highlight a LotusScript keyword and press **F1** to display a Help topic about the keyword.

Entering statements in your script editor

A script is composed of one or more statements in the LotusScript language. Each **statement** can consist of keywords, operators, literals, identifiers, and punctuation, written according to the syntax rules for that statement.

The following illustration shows several ways to enter statements in your script editor:

```
Print "One line"              ' One statement on one line.

Print  "One" & _              ' One statement on two lines.
   "Two"

Print "One" : Print "Two"     ' Two statements on one line.
```

Keep the following in mind when you enter statements in the Script Editor.

- Enter statements as lines of text. Each text element is a LotusScript keyword, operator, identifier, literal, or special character.
- Scripts can include blank lines.
- You can enter text at the left margin or indented without affecting its meaning.
- Separate text elements with white space such as spaces or tabs. Extra white space helps make statements more readable, but it has no effect.
- Statements typically appear one to a line. A newline marks the end of a statement, except for a **block statement**, which appears on multiple lines and is delimited by two or more keywords. The beginning of the next line starts another statement.

- Use an underscore (_) to continue a statement to the next line. Precede the underscore character (_) by white space. Only white space or same-line comments (those preceded with an apostrophe) can follow the underscore on the line. You cannot continue a line within a literal string or a comment.
- Separate multiple statements on a line with a colon (:), except in the IDE.

Entering numbers

Enter numbers in scripts according to the rules in the following table:

Kind of literal number	*Example*	*Constructing numbers*
Decimal integer	777	The legal range is the range for Long values. If the number falls within the range for Integer values, its data type is Integer; otherwise, its data type is Long.
Decimal	7.77	The legal range is the range for Double values. The number's data type is Double.
Scientific notation	7.77E+02	The legal range is the range for Double values. The number's data type is Double.
Binary integer (base 2)	&B1100101	The legal range is the range for Long values. A binary integer can have 32 binary digits of 0 or 1. Values of &B100000 ... (31 zeroes) and larger represent negative numbers.
Octal integer (base 8)	&O1411	The legal range is the range for Long values. An octal integer can have 11 octal digits of 0 to 7. Values of &O20000000000 and larger represent negative numbers. Values of &O40000000000 and larger are out of range.
Hexadecimal integer (base 16)	&H309	The legal range is the range for Long values. A hexadecimal integer can have eight hexadecimal digits of 0 to 9 and A to F. Values of &H80000000 and larger represent negative numbers.

Entering strings

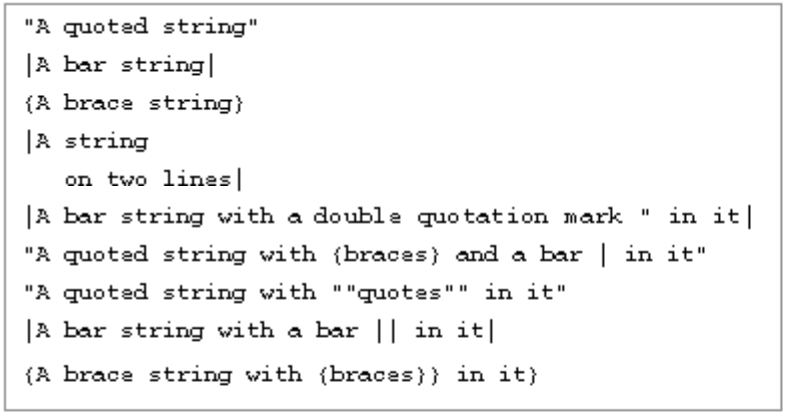
A **string** is a set of characters enclosed in quotation marks (""), vertical bars (| |), or open and close braces ({ }).

Keep the following in mind when you enter strings in a script:

- Strings enclosed in vertical bars or braces can span multiple lines.
- To enter a closing delimiter character ", |, or } as text in a string delimited by that character, enter the character twice. To enter the { character as text in a string, enter it once. For example, the statement **`Print {foo{8}}`** prints foo{8}.

- Enter "" to specify the empty string.
- You cannot nest strings that are delimited by vertical bars, braces, or double quotation marks.

The following illustration shows some examples of strings.



```
"A quoted string"
|A bar string|
{A brace string}
|A string
   on two lines|
|A bar string with a double quotation mark " in it|
"A quoted string with {braces} and a bar | in it"
"A quoted string with ""quotes"" in it"
|A bar string with a bar || in it|
{A brace string with {braces}} in it}
```

Entering identifiers

An **identifier** is the name you give to a variable, a constant, a type, a class, a sub, or a property. An identifier can also be used as a label if you omit the data type suffix character.

Here's how to construct an identifier:

- The first character in an identifier must be a letter (upper or lower case).
- The remaining characters can be letters, digits, or the underscore character.
- You can append a data type suffix character (one of the characters %, &, !, #, @, and $) to an identifier.
- The maximum length of an identifier is 40 characters, excluding the optional suffix character.
- Identifiers are case-insensitive. For example, MyVariable is the same name as myvariable.
- Identifiers can have characters with ANSI codes above 127 (that is, characters outside the ASCII range).

Escape character for illegal names

Some Lotus-product classes and OLE classes define properties or methods whose identifiers contain illegal LotusScript characters. Lotus-product variable names may also include illegal LotusScript characters. In these cases, prefix the illegal character with a tilde (~), the Escape character, to prevent error messages. For example:

```
Call ProductClass.$MyMethod      ' Illegal
Call ProductClass.~$MyMethod     ' Legal
```

Entering labels

A **label** gives a name to a statement. You can use the following statements to transfer control to a labeled statement by referring to its label:

```
GoSub...Return
GoTo
If...GoTo...Else
On Error
On...GoSub...Return
On...GoTo
Resume
```

The following rules apply to constructing labels:

- A label can be any name (up to 40 characters).
- A label can appear only at the beginning of a line; it labels the first statement on the line.
- A label can appear on a line by itself; this labels the first statement following the line.
- A statement can have more than one label preceding it; but the labels must appear on different lines.
- A given label cannot label more than one statement in the same procedure.
- A label cannot include a data type suffix character.

Entering keywords

A **keyword** is a word with a fixed spelling and a particular meaning in the LotusScript language. It is distinguished from an identifier, a word whose spelling and meaning you choose. The keywords name LotusScript statements, built-in functions, predefined constants, and data types. Keywords are “reserved words”: their meaning is specified by LotusScript, and they cannot be used with any other meaning in a script. The one exception to this rule is that most keywords can be used as names to name variables within a user-defined data type, and variables and methods within a class.

For a list of LotusScript keywords, see the *LotusScript Language Reference*.

Entering special characters

You use special characters, such as punctuation marks, to delimit literal strings, designate variables as having particular data types, punctuate lists such as argument lists and subscript lists, punctuate statements, and punctuate lines in a script.

The following table shows LotusScript special characters and describes their usage:

Character	*Description*
" (quotation mark)	Opening and closing delimiter for a string on a single line.
| (vertical bar)	Opening and closing delimiter for a multiline string. To include a vertical bar in the string, use double bars (| |).
{ } (braces)	Delimits a multiline literal string. To include an open brace in the string, use a single open brace ({). To include a close brace in the string, use double close braces (}}).
: (colon)	Separates multiple statements on a line, except in the IDE. When following an identifier at the beginning of a line, designates the identifier as a label.
$ (dollar sign)	When suffixed to the identifier in a variable declaration, declares the data type of the variable as String. When prefixed to an identifier, designates the identifier as a product constant.
% (percent sign)	When suffixed to the identifier in a variable declaration, declares the data type of the variable as Integer. When suffixed to either the identifier or the value being assigned in a constant declaration, declares the constant's data type as Integer. Also, designates a compiler directive, such as %Rem.
& (ampersand)	When suffixed to the identifier in a variable declaration, declares the data type of the variable as Long. When suffixed to the identifier or assigned value in a constant declaration, declares the constant's data type as Long. Also, prefixes a binary (&B), octal (&O), or hexadecimal (&H) number, or designates the string concatenation operator in an expression.
! (exclamation point)	When suffixed to the identifier in a variable declaration, declares the data type of the variable as Single. When suffixed to the identifier or assigned value in a constant declaration, declares the constant's data type as Single.
# (pound sign)	When suffixed to the identifier in a variable declaration, declares the data type of the variable as Double. When suffixed to the identifier or the assigned value in a constant declaration, declares the constant's data type as Double. When prefixed to a literal number or a variable identifier, specifies a file number in certain file I/O statements and functions.
@ (at sign)	When suffixed to the identifier in a variable declaration, declares the data type of the variable as Currency. When suffixed to the identifier or the assigned value in a constant declaration, declares the constant's data type as Currency.
***** (asterisk)	Specifies the string length in a fixed-length string declaration, or designates the multiplication operator in an expression.

Continued

Character	*Description*
( ) (parentheses)	Groups an expression, controlling the order of evaluation of items in the expression. Encloses an argument in a sub or function call that should be passed by value. Encloses the argument list in function and sub definitions, and in calls to functions and subs. Encloses the array bounds in array declarations, and the subscripts in references to array elements. Encloses the list tag in a reference to a list element.
. (period)	In dot notation for a user-defined data type variable or an object reference variable, references members of the type or object. In dot notation for a product object reference, designates the selected product object. In dot notation for an object reference within a With statement, designates the object referred to by the statement. Also, designates the decimal point in a floating-point literal value.
.. (two periods)	Within a reference to a procedure in a derived class that overrides a procedure of the same name in a base class, specifies the overridden procedure.
[ ] (brackets)	Delimits names used by certain Lotus products to identify product names.
, (comma)	Separates arguments in calls to functions and subs, and in function and sub definitions. Separates bounds in array declarations, and subscripts in references to array elements. Separates expressions in Print and Print # statements. Separates elements in many other statements.
; (semicolon)	Separates expressions in a Print statement or a Print # statement.
' (quote)	Indicates the beginning of a comment. The comment continues to the end of the line on which the comment begins.
_ (underscore)	Continues a line of code from one line to the next when preceded by at least one space or tab.

Use white space to separate names and keywords, or to make the use of a special character unambiguous. Avoid using white space around a special character, such as a data type suffix character, appended to a name.

If you want to use special characters as ordinary text characters, enter the special characters in strings.

Compiling Scripts

An application must be compiled before it will load and execute. Both the uncompiled and compiled versions of an application file have an extension that is specific to your Lotus product.

When you compile a script, LotusScript displays messages about any errors it finds, listed in the order in which they are found. A **compile-time error** occurs when a script contains an error that LotusScript detects during compilation. For example, an If...End

If statement might be missing its required End If clause, or a keyword might be misspelled. Another type of compile-time error is the **syntax error**, where LotusScript punctuation or grammar is used incorrectly. Examples of syntax errors include forgetting to match parentheses or forgetting to terminate a string with a quotation mark.

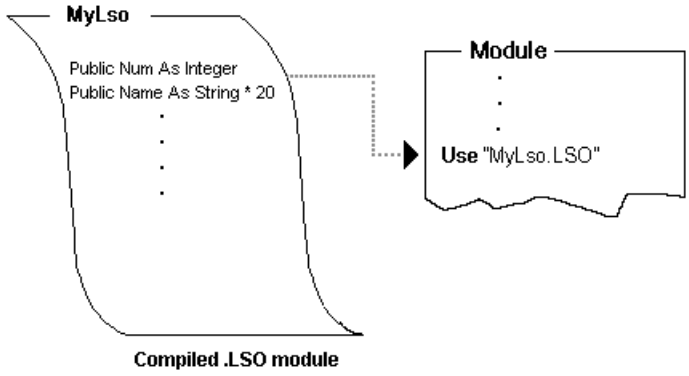
As you fix errors, you can recompile until there are no more errors in the script. You can compile your scripts explicitly, using your product's menu commands, or you can let the compiler compile them automatically when you save the application or when you run it. For information about whether your product allows you to compile scripts explicitly or implicitly, see the product documentation.

Creating and using compiled script modules

Some Lotus products allow you to write and compile script modules as files with an .LSO extension and then use these files in your applications. This feature lets you create one copy of a compiled script module to use in multiple applications.

You can use your script editor, or any text editor, to create the script. The script can contain declarations, subs, and functions, including the definition and declarations of product classes, properties, subs, and functions.

To use a compiled LotusScript module, put a LotusScript Use statement in a script that includes declarations. For more information, see the product documentation.



If you place the Use statement in a declarations section, any public declarations, subs, and procedures in the .LSO file are available to all the scripts in the corresponding module. If your Lotus product provides a Public script, place the Use statement in this script to make Public declarations and procedures in the .LSO file available to all scripts in the application.

If you change the file name or file extension of the .LSO file after you compile it, LotusScript will not be able to use the script module. The original file name is embedded in the compiled module. To change the file name, you must rename the source file and compile the .LSO file.

Debugging Your Application

Your debugger helps you find errors in the logic of your application. If your application compiles without errors but does not yield the results you expect, your debugger can help you locate the place in your scripts where something went wrong.

When you run an application with your debugger, the application is in one of three states:

- Running: Application scripts run uninterrupted until LotusScript reaches a breakpoint or Stop statement. A **breakpoint** is a statement at which you want to interrupt application execution.
- Interrupted: LotusScript interrupts application execution and passes control to your debugger.
- Stepping: LotusScript passes control to the application scripts and then back to your debugger after executing a single statement in a script.

When you debug an application, some Lotus products allow you to inspect variables and edit their values. For more information, see the product documentation.

Chapter 3
Data Types, Constants, and Variables

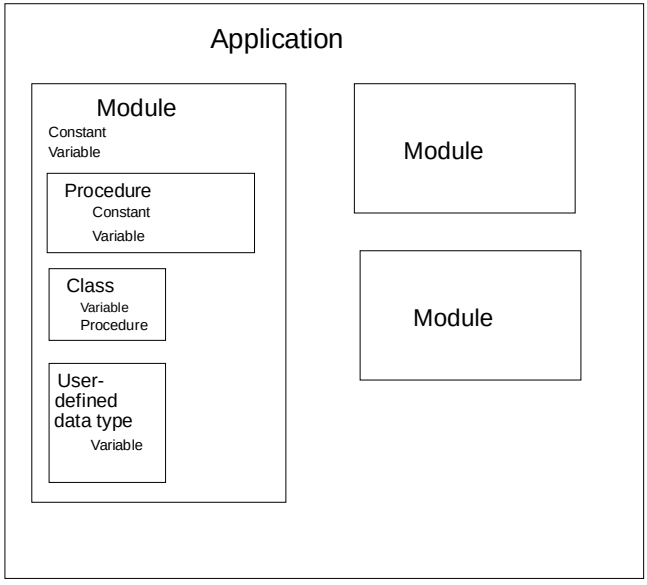
This chapter provides information about LotusScript constants and variables and the data types of the values that they can represent. It covers the following topics:

- Overview of constants and variables, scope and lifetime
- Summary of the data types that LotusScript recognizes
- Constants: built-in and user-defined constants
- Variables: the scalar types, explicit and implicit declarations, and a sampling of the LotusScript functions that apply to scalars
- Arrays: fixed and dynamic arrays
- Lists
- Variants: declaring and referring to variables of type Variant, using Variants to manipulate Boolean and date/time values
- Data type conversion: explicit and automatic data type conversion

A LotusScript application can manipulate data of several types through the use of constants and variables. **Constants** and **variables** are identifiers that name locations in memory that hold (or can hold) data of one or another of the types that LotusScript recognizes. (It is common shorthand to say that a constant or variable "holds" or "has" such and such a value when what is actually meant is that the constant or variable names the location that contains the value in question.) Constants differ from variables in that the value that a constant represents must be known at compile time and can't be changed—it must remain constant—while the application is running, while a variable can refer to a value (or a set of values) that can change while the application is running.

Like other identifiers, constants and variables have a scope and a lifetime. **Scope** refers to the area of an application in which an identifier can be referred to, that is, the area in which the identifier is accessible, or known. **Lifetime** (or **persistence**) refers to the period during which the identifier is available to the application. When you define a constant or declare a variable, LotusScript assigns it a default scope and lifetime, which in some cases you can override by including the appropriate keyword in the definition or declaration.

The specific areas of an application in which a constant or variable (or any other identifier) is known, and for what duration, depend on the application model that a product and its programming environment support. The following diagram shows the generic application model assumed throughout this book and the areas in which you can define constants and declare variables:



Summary of LotusScript Data Types

LotusScript recognizes the following numeric and string data types (called **scalar data types**):

Data type	*Value range*	*Size*
Integer Signed short integer	-32,768 to 32,767	2 bytes
Long Signed long integer	-2,147,483,648 to 2,147,483,647	4 bytes
Single Single-precision floating-point	-3.402823E+38 to 3.402823E+38	4 bytes
Double Double-precision floating-point	-1.7976931348623158+308 to 1.7976931348623158+308	8 bytes

Continued

Data type	*Value range*	*Size*
Currency Fixed-point integer scaled to 4 decimal places	-922,337,203,685,477.5807 to 922,337,203,685,477.5807	8 bytes
String	0 to 32K characters (0 to 64K bytes)	2 bytes / character

Besides these scalar data types, LotusScript supports the following additional data types and data structures:

Data Type or structure	*Description*	*Size*
Array	A set of elements having the same data type. An array can comprise up to 8 dimensions whose subscript bounds can range from -32,768 to 32,767.	up to 64K bytes
List	A one-dimensional set whose elements have the same data type and are referred to by name rather than by subscript.	up to 64K bytes
Variant	A special data type that can contain a value of any scalar value, array, list, or object reference. Variants can also hold Boolean and date/time values.	16 bytes
User-defined data type	A set of elements of possibly disparate data types. Comparable to a record in Pascal or a struct in C.	up to 64K bytes
User-defined class	A set of elements of possibly disparate data types together with procedures that operate on them.	
Object reference	A pointer to an OLE Automation object or an instance of a product-defined class or user-defined class.	4 bytes

Arrays, lists, and Variants are described in detail later in this chapter. For more information about user-defined data types, user-defined classes, and object references, see Chapter 5, “Creating User-Defined Data Types and Classes.”

Constants

Strictly speaking, a **constant** names a location in memory that contains a value that is known at compile time and cannot be changed while the application is running. In less formal terms, a constant is a named fixed value. Constants are defined in the following ways:

- By LotusScript, internally. These constants are built into the language and are always available to an application.
- By LotusScript, in the file LSCONST.LSS. These constants are available in a module only when the module explicitly includes the file in which they are defined.
- By an individual product, in a file that that product makes available. The file in which these constants are defined may or may not have to be included explicitly in the module in which you want to use them.
- By the application developer, in an application module or in a file that you explicitly include in a module.

Built-in constants

LotusScript provides an internal value named EMPTY, and five other constants with predefined values, which are summarized in the following table:

Constant	*Value*
EMPTY	The initial value of a Variant variable. LotusScript converts EMPTY to the empty string ("") in string operations and to 0 in numeric operations. To test a variable for the EMPTY value, use the IsEmpty function. You cannot assign EMPTY as a value.
NOTHING	The initial value of an object reference variable. As soon as you assign a specific reference to the variable, the variable no longer contains NOTHING. You can explicitly assign the value NOTHING to an object reference variable. To test a variable for the NOTHING value, use the Is operator.
NULL	A special value that represents unknown or missing data. Various operations return a NULL value, but you can only assign the NULL value to a Variant variable. To determine if a variable contains the NULL value, use the IsNull function.
PI	The ratio of the circumference of a circle to its diameter. This constant can be assigned to any numeric variable, or used in numeric expressions.
TRUE and FALSE	The Boolean values True and False, which LotusScript evaluates as the integer values -1 and 0, respectively. These values are returned by all comparison and logical operations. In an If, Do, or While statement, which test for TRUE or FALSE, any nonzero value is considered True.

Constants defined in LSCONST.LSS

LotusScript provides a set of constants that you can use in place of numeric arguments in certain LotusScript statements, such as MessageBox:

```
' Declare an Integer variable, theStr%,
' and assign it to the sum of two Integer constants.
Dim theStr%
theStr% = MB_YESNO + MB_ICONQUESTION
MessageBox "Do you want to continue?", theStr%, "Continue?
```

which is much more readable than

```
MessageBox "Do you want to continue?", 4 + 32, "Continue?
```

These constants are defined in the file LSCONST.LSS. Use the %Include directive to incorporate this file into your application in a module that must be loaded when you need to use the constants, which are all explicitly defined to be Public. The syntax for including this file is:

```
%Include "LSCONST.LSS"
```

Product-specific constants

Individual Lotus products may provide additional constants that you can use by including the file in which they are defined in your application with the %Include directive. A product may also provide internally defined constants that are automatically available to your application. Consult the product's documentation to learn about these constants.

User-defined constants

You can define your own constants within a module or a procedure. Such a constant can be of any of the scalar data types that LotusScript recognizes. Use the following syntax to define a constant:

[**Public** | **Private**] **Const** *constName = expression*

The syntax elements in the definition of a constant are summarized in the following table:

Element	*Description*
Public, Private	Only an option when you declare a constant at module level, not within a procedure. Public means that the constant can be used outside the module in which it is defined. Private means the constant can only be used inside the module in which it is defined. Constants are Private by default.
constName	The name of the constant. The name, which can include a data type suffix character, must be a legal LotusScript identifier (see Chapter 2). A constant cannot have the same name as another constant, variable, or procedure of the same scope used in the same module.
expression	An expression indicating the value of the constant. The expression can be a literal or another constant. You can use arithmetic and logical operators in the expression. The expression can contain a LotusScript function (such as Abs or UCase$) if that function can be evaluated at compile time and its arguments (if any) are constant.

You can define a constant to be of a particular data type by appending one or another of the following data type suffix characters to *constName*:

Suffix	*Data type*
%	Integer
&	Long
!	Single
#	Double
@	Currency
$	String

For example:

```
' Define a String constant, MYNAME.
Const MYNAME$ = "Andrea"
' Define a Single constant, MYPERCENT.
Const MYPERCENT! = 0.125
' Define a Currency constant, MYMONEY.
Const MYMONEY@ = 123.45
```

Alternatively, if the constant is numeric, and *expression* is a numeric literal, you can specify the particular numeric data type by appending the appropriate data type suffix character to *expression*. For example:

```
' Define a Currency constant, MYCUR, with the value 123.45.
Const MYCUR = 123.45@
```

If you don't append a suffix character to *constName* or *expression*, LotusScript determines the data type of the constant by the value of *expression*.

- For a string, the data type is String.
- For a Single or Double value, the data type is Double.
- For an integer, the data type is Integer or Long, depending on the magnitude of the value.

For example:

```
Const MYNAME = "Sara"
' MYNAME is a constant of type String.
Const MYDOUBLE = 123.45
' MYDOUBLE is a constant of type Double.
Const MYINT = 123
' MYINT is an constant of type Integer.
Const MYLONG = 123456
' MYLONG is a constant of type Long.
```

You can always include a data type suffix character when you refer to a constant in a LotusScript application, whether or not you used the suffix character in the Const statement that defined the constant. You need not use the suffix, though it makes your code easier to read. But if you do, make sure that the suffix is the appropriate one.

For example:

```
Const MYADDRESS$ = "722 Smith Place"
Print MYADDRESS
' Output: 722 Smith Place

Const YOURADDRESS = "75 rue St. Viateur"
Print YOURADDRESS$
' Output: 75 rue St. Viateur
' Print MYADDRESS%, YOURADDRESS@ would cause an error.
```

Testing for the data type of a constant

You can ascertain a constant's data type by calling either of two LotusScript functions: TypeName and DataType. TypeName returns a string indicating the data type of the expression being tested, and DataType returns a number representing the expression's data type. (For a complete listing of the values that TypeName and DataType return, see the *LotusScript Language Reference* or online Help.) For example:

```
Const MYMONEY@ = 123.45
Const MOREMONEY = MYMONEY * 2
Print TypeName(MOREMONEY)
' Output: CURRENCY
Print DataType(MOREMONEY)
' Output: 6
```

The scope of a constant

You can define a constant within a procedure or at module level (that is, outside the definition of a procedure, user-defined data type, or class). A constant that you define within a procedure is accessible only within that procedure though the procedure itself may be available to the whole module or application. If that constant has the same name as a constant or variable defined outside the procedure, LotusScript interprets references inside the procedure to that name as applying to the constant with the narrower scope, ignoring the existence of the constant with the greater scope. For example:

```
Const MYINT% = 10
' This MYINT% is defined at module level.
Sub MySub
   Const MYINT% = 100
   ' This MYINT% is defined within a procedure.
   Print MYINT%
End Sub
Call MySub
' Output: 100
Print MYINT%
' Output: 10
```

By default, a constant that you define at module level is Private, that is, accessible only within that module. You can override this default in either of two ways to make the constant available to other modules in the application:

- Include the keyword Public in the statement that defines the constant, for example:

  ```
  Public Const GLOBALINT% = 123
  ```

- Include the Option Public statement at the beginning of a module that must be loaded when the application runs. This makes all identifiers in the module Public by default.

To access a Public constant defined in another module, you compile that module and then refer to the compiled module in a Use statement in the accessing module. (This is how you access any item defined as Public, whether a constant, variable, procedure, user-defined data type definition, or class definition.) For example, to access the Public constants in module A from module B, you compile module A and then include the following statement in module B:

```
Use "A"
```

Variables

Like a constant, a **variable** names an area of storage. Unlike a constant, however, the value assigned to a variable can change during execution of an application.

You declare a variable to be of a particular type, which restricts the kind of value the variable can hold. You also determine the scope and lifetime of a variable—when and how long the variable exists and in what parts of your application it is accessible. Typically, if you do not choose a particular type or scope for the variable, LotusScript chooses a type and scope by default.

A variable name can be any valid LotusScript identifier. The name cannot be the same as the name of another variable, constant, or procedure in the same scope used in the same module.

A variable can be of any of the following data types or structures:

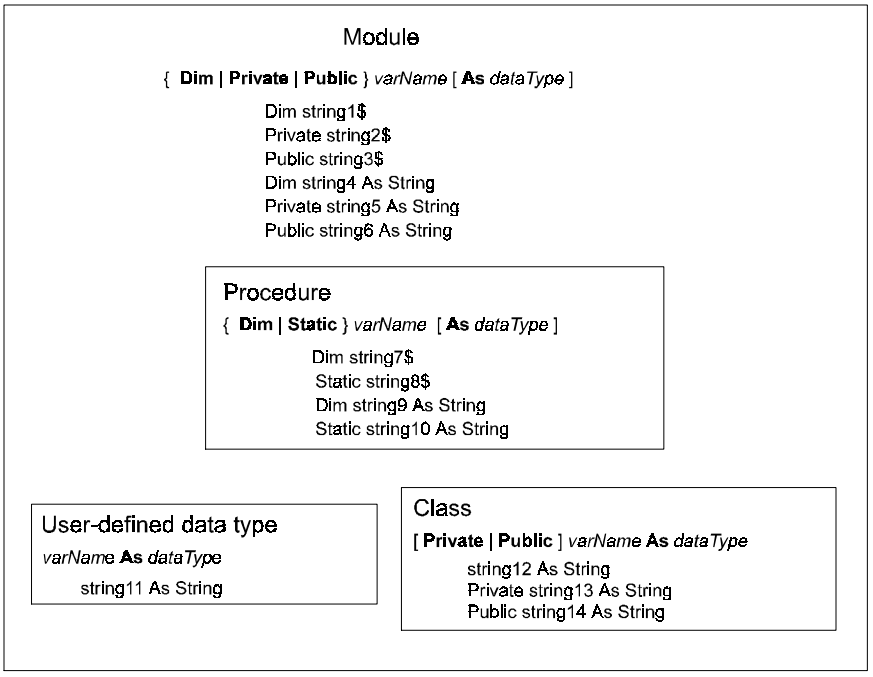
- One of the scalar types that LotusScript recognizes: Integer, Long, Single, Double, Currency, or String
- An array or a list
- A Variant
- A user-defined data type, that is, a type defined with a Type...End Type statement
- A class, that is, a class defined with a Class...End Class statement, or a class defined by the Lotus product with which LotusScript is running

The next two sections of this chapter describe the two ways you can declare a scalar variable in LotusScript: with an explicit statement or by implication. Subsequent sections describe how to declare arrays, lists, and variables of type Variant. For more information about user-defined data types and classes and the variables that are associated with them, see Chapter 5 (“Creating User-Defined Data Types and Classes”).

Declaring scalar variables explicitly

Declaring a variable creates an identifier, determines its scope and lifetime, specifies the type of data that can occupy the location in memory to which it refers, and causes LotusScript to write an initial value to that location. The recommended way to accomplish all this is to declare the variable explicitly. You declare a scalar variable explicitly with the Dim statement, or one of its variations. The variation you use depends in part on the area of the application in which you declare the variable, and in part on the scope and lifetime you want the variable to have.

The following diagram summarizes the syntax for declaring a single scalar variable (in this example, a variable of type String):



The syntax elements in the declaration of a scalar variable are summarized in the following table:

Element	*Description*
Dim	Declares a variable with Private scope.
Public, Private	Public declares a variable with Public scope. Private declares a variable with Private scope.
Static	Only applicable to variables declared inside a procedure. Static variables retain their values (rather than going out of existence) between calls to the procedure while the module in which the procedure is defined remains loaded.
varName	The name of the variable. At module level or within a procedure, *varName* can end in one or another of the data type suffix characters that LotusScript recognizes as such. This determines the type of data that the variable can hold. You can append a data type suffix character to a variable name when you declare it only if you do not include the As *dataType* clause in the declaration.

Continued

Element	*Description*
As *dataType*	Specifies the type of data the variable can hold. If you include this clause, *varName* cannot end in a data type suffix character. This clause is required in the declaration of a variable within the definition of a user-defined data type or class, but optional in the declaration of a variable at module level or within a procedure.

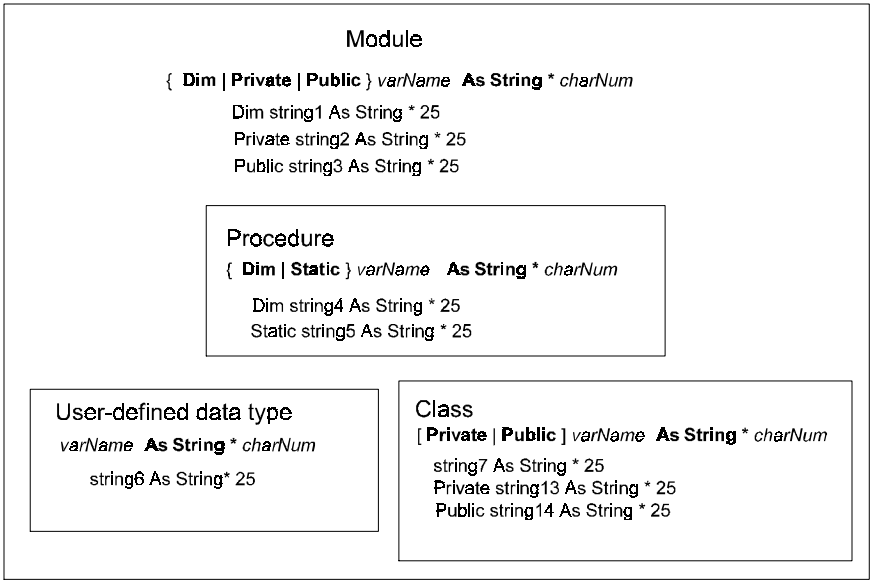
Whether or not you append a data type suffix character to the name of the variable when you declare it, you can always do so (or not) when referring to an explicitly declared scalar variable. For example:

```
Public firstName$
Public lastName As String
Dim age%
Dim money As Currency

firstName$ = "Roman"
lastName$ = "Minsky"
age% = 12
money@ = 150.75
Print firstName & " " & lastName & ", " & age &", $" & money
' Output: Roman Minsky, 12, $150.75
Print firstName$ & " " & lastName$ & ", " & age% &", $" & money
' Output: Roman Minsky, 12, $150.75
```

String variables

A variable of type String contains a sequence of characters in the **Unicode** character set. Unicode is a character-encoding system that uses two bytes to represent each character in the set. LotusScript converts input to Unicode format before compiling an application.

A String variable can be of variable or fixed length. The syntax for declaring a variable-length String variable is shown in the preceding diagram. The syntax for declaring a fixed-length String variable is shown below:



The *charNum* argument specifies that *varName* is a fixed-length String variable of *charNum* characters.

When you assign a string to a fixed-length String variable, LotusScript truncates the string or pads it to the declared length with trailing spaces if necessary.

For example:

```
Dim myName$
Dim myTown As String
' myName and myTown are variable-length string variables.
Dim myState As String * 2
' myState is a 2-character fixed-length String variable.
Dim myZIP As String * 5
' myZIP$ is a 5-character fixed-length String variable.
' If myZIP$ is assigned a value of more than 5 characters,
' that value will be truncated to its first 5 characters.
myName$ = "Mark"
myTown$ = "Centerville"
myState$ = "MA"
myZIP$ = "02100-9999"
Print myName$
' Output: Mark
Print myTown$ & ", " & myState$ & " " & myZIP$
' Output: Centerville, MA 02100
```

Declaring more than one variable at a time

The Dim statement and its variations allow you to declare more than one variable at a time at module level or within a procedure. At module level, the syntax is

{ **Dim** | **Public** | **Private** } *varName1* [**As** *dataType*]**,** *varName2* [**As** *dataType*]**,** ...

Within a procedure, the syntax is

{ **Dim** | **Static** } *varName1* [**As** *dataType*]**,** *varName2* [**As** *dataType*]**,** ...

The conventions for appending a data type suffix character to a variable name in the absence of an As *dataType* clause (and not appending a data type suffix character in the presence of an As *dataType* clause) are the same as in the declaration of a single scalar variable. For example:

```
Dim aString$, anInt%, aDouble As Double, aCurrency@
aString$ = "Hello"
Print TypeName(aString$) & ": " & aString$
' Output: STRING: Hello
anInt% = 123
Print TypeName(anInt%) & ": " & anInt%
' Output: INTEGER: 123
aDouble# = 123.45
Print TypeName(aDouble) & ": " & aDouble#
' Output: DOUBLE: 123.45
aCurrency@ = 456.78
Print TypeName(aCurrency@) & ": " & aCurrency@
' Output: CURRENCY: 456.78
```

```
Sub MySub
  Dim aString As String * 2, anotherString$, anInt%
  Static aDouble#, anotherDouble#

  aString$ = "Hi"
  Print TypeName(astring$) & ": " & aString$
  anotherString$ = "World"
  Print TypeName(anotherstring$) & ": " & anotherString$
  anInt% = 234
  Print TypeName(anInt%) & ": " & anInt%
  aDouble# = aDouble# + 1
  anotherDouble# = aDouble# * 2
  Print TypeName(anotherDouble#) & ": " & anotherDouble#
End Sub
Call MySub
' Output:
' STRING: Hi
' STRING: World
' INTEGER: 234
' DOUBLE: 2
Call MySub
' Output:
' STRING: Hi
' STRING: World
' INTEGER: 234
' DOUBLE: 4
```

Initial default values

When you declare a variable explicitly, LotusScript assigns it an initial default value:

Type of variable	*Initial value*
Numeric (Integer, Long, Single, Double, Currency)	0
Variable-length String	"" (the empty string)
Fixed-length String	A string of the specified length, filled with Chr(0) (the NULL character)

Declaring scalar variables implicitly

At module level or within a procedure, you can declare a variable implicitly by assigning a value to an identifier that you have not previously declared, as in the following example:

```
' Create an Integer variable without declaring it explicitly
' and initialize it to 1.
counter% = 1
```

This has the same effect as the following explicit declaration and statement:

```
Dim counter%
counter% = 1
```

As with explicitly declared variables, the identifier has to be a legal one and not already in use as the name of a constant, variable, or procedure in the same scope in the same module. If you append a data type suffix character to the variable's name when you declare it, that suffix determines the variable's data type. If you don't append a data type suffix character, one of two things happens: if the name begins with a character covered by an existing Def*type* statement (see below), the variable is implicitly declared to be of the data type appropriate to that statement. Otherwise, the variable is implicitly declared to be of type Variant. The same rules apply to explicitly declared variables if the declaration doesn't contain an As *dataType* clause and the variable name doesn't end in a data type suffix character:

```
' Declare a variable of type Variant.
Dim myVarV
```

Implicit declaration is a handy shortcut when you're writing a simple script, saving you the line of code that it would take to declare the variable explicitly. However, the line of code you save by collapsing the declaration of a variable and the assignment of a value into a single statement can be costly in an application of even moderate complexity for two reasons:

- When you implicitly declare a variable of one of the scalar types by including the appropriate data type suffix character, LotusScript requires you to use that character whenever you subsequently refer to that variable. Omitting the data type suffix character in referring to such a variable produces an error. The opposite is true of implicitly declared variables covered by Def*type* statements: they are declared without a data type suffix character, and you can't include one when you refer to them later in the application without producing an error.
- If you omit the data type suffix character in an implicit declaration and the identifier isn't covered by an existing Def*type* statement, you implicitly declare a variable of type Variant, which is not necessarily what you want to do. While useful in many ways, Variants take up more storage space in memory than the other scalar types. And if you include a data type suffix character when referring to a variable of type Variant, this produces an error.

For example:

```
' Create the Integer variable anInt without explicitly
' declaring it and initialize it to 10.
anInt% = 10
Print anInt
' Produce "Name previously declared" error
' because LotusScript reads anInt (without suffix character)
' as an implicitly declared Variant variable, not
' the Integer variable anInt% (with suffix character).

' Create the Variant variable myVariantV without explicitly
' declaring it and initialize it to 10.
myVariantV = 10
Print myVariantV%
' Produce "Type suffix mismatch" error
' because myVariantV (without suffix character) was declared
' as type Variant, but the suffix character % is only appropriate
' for variables declared as type Integer.
```

If you want to disallow implicit declaration in a LotusScript application, you can do so by including the Option Declare statement at module level in a module that you plan to have loaded when the application runs. This statement tells LotusScript to require explicit declarations for all your variables.

Def*type* Statements

You use a LotusScript Def*type* statement at module level to assign a default data type to variables whose names begin with a particular letter of the alphabet, don't end with a data type suffix character, and don't appear in an explicit declaration containing an As *dataType* clause. The statement must appear before any variables are declared in the module. The syntax is

Def*type range* [, *range*]...

where *type* is a suffix such as Cur or Dbl, which is an abbreviation of the name of a data type, and *range* is one or more consecutive letters of the alphabet. For example:

```
' Implicitly declared variables beginning with
' A, a, B, b, C, or c will be of type Integer.
DefInt A-C
' Create the Integer variable anInt on the fly
' and initialize it to 10.
anInt = 10
' Create a variable of type Variant on the fly
' and initialize it to 2. It's a Variant because
' it doesn't have a data type suffix character and doesn't
' begin with any of the characters in the specified
' DefInt range.
smallIntV = 2
```

More about scalar variables

LotusScript provides a set of built-in functions that enable you to manipulate scalar values in various ways. A **built-in function** is a named procedure that is part of the LotusScript language and typically performs some operation on a value that you pass it, producing a new value, called the **return value**. Most of these functions fall into one or another of the following four categories:

- Numeric
- String
- Date/time
- Data type conversion

See the *LotusScript Language Reference* or online Help for a complete list and detailed description of these functions.

The following examples contain a representative sampling of the LotusScript numeric and string functions and illustrate some of the things you can do with them. Each example is a Print statement, which causes LotusScript to display the return value of the particular function.

```
Dim anInt As Integer
Dim aDouble As Double
aDouble# = -123.654
anInt% = 6

' Ascertain if aDouble# is a numeric
' data type: True (-1) or False (o).
Print IsNumeric(aDouble#)
' Output: True

' Ascertain if anInt% is positive (1),
' negative (-1), or neither (0).
Print Sgn(anInt%)
' Output: 1

' Print the absolute value of aDouble#.
Print Abs(aDouble#)
' Output: 123.654

' Print aDouble# rounded to 1 decimal place.
Print Round(aDouble#,1)
' Output: 123.7

' Print the nearest integer equal to or less than aDouble#.
Print Int(aDouble#)
' Output: -124
```

```
' Print the integer part of aDouble#.
Print Fix(aDouble#)
' Output: -123

' Print the decimal part of aDouble#.
Print Fraction(aDouble#)
' Output: -.653999999999996

' Print the exponential (base e) of anInt%.
Print Exp(anInt%)
' Output: 403.428793492735

' Print a random whole number between 1 and 5
' by seeding the random number generator,
' calling the Rnd function to generate a random number,
' and performing various operations on the result.
' First, seed the random number generator.
Randomize
' Generate a random decimal number;
' take its decimal part and round it to one decimal place;
' multiply the result by 10 to make it a one-digit whole number;
' divide that number by 5 and add 1 to the remainder. The result
' is a random whole number between 1 and 5.
Print ((round(Fraction(Rnd),1) * 10) Mod 5) + 1
' Output: a random integer between 1 and 5.

Dim aString As String
Dim theNewString As String

' Assign aString the value (space) (space) abcdef (space) (space).
aString$ = chr$(32) + chr$(32) + "abcdef" + chr$(32) + chr$(32)
Print aString$
' Output: (space) (space) abcdef (space) (space)

' Ascertain the number of characters that aString$ contains.
Print Len(aString$)
' Output: 10

' Strip leading and trailing spaces from aString$.
aString$ = Trim$(aString$)
Print aString$
' Output: abcdef
Print Len(aString$)
' Output: 6

' Convert all the alphabetic characters in aString$ to uppercase.
aString$ = UCase$(aString$)
Print aString$
' Output: ABCDEF
```

```
' Print the leftmost 3 characters of aString$.
Print Right$(aString$, 3)
' Output: ABC

' Print the position in aString$ where the substring "DE" begins.
Print InStr(aString$, "DE")
' Output: 4

' Print the first two characters of the substring that starts
' at the fourth character of aString$.
Print Mid$(aString$,4, 2)
' Output: DE

' Assign theNewString$ a value of a string of 10 asterisks.
theNewString$ = String$(10, "*")
Print theNewString$
' Output: **********

' Starting at the third character of aString$, replace the next
' 2 characters of aString$ with the first 2 characters of
' theNewString$.
Mid$(aString$,3,2 ) = theNewString$
Print aString$
' Output: AB**EF
```

Arrays

An **array** is a named collection of elements of the same data type, where each element can be accessed individually by its position within the collection. If a scalar variable names a single location in memory, an array variable names a series of locations in memory, each holding a value of the same type—Integer or String, for example.

The position of an element in an array can be identified by one or more coordinates called **subscripts** (or **indexes**). The number of subscripts necessary to identify an element is equal to the number of the array's **dimensions**. In a one-dimensional array, a given element's position can be described by one subscript; in a two-dimensional array, it takes two subscripts to locate an element; in a three-dimensional array, it takes three subscripts, and so on.

For example, in a one-dimensional array whose elements are the names of the states of the United States, a single subscript suffices to identify the position of a given state in the collection:

```
Dim states(1 to 50) As String
states(1) = "Alabama"
states(2) = "Alaska"
states(3) = "Arizona"
' and so on.
Print states(2)
' Output: Alaska
```

In a two-dimensional array whose elements are the names of the ten most populous cities in each state, two subscripts suffice to identify the position of a given city in a given state: one of these identifies the state, and the other identifies the city:

```
Dim statesAnd10Cities(1 to 50, 1 to 10) As String
statesAnd10Cities(1,1) = "Alabama, Birmingham"
statesAnd10Cities(1,2) = "Alabama, Mobile"
' ...
statesAnd10Cities(2,1) = "Alaska, Anchorage"
statesAnd10Cities(2,2) = "Alaska, Fairbanks"
' and so on.
Print statesAnd10Cities(1,2)
' Output: Alabama, Montgomery
```

A three-dimensional array might contain the numbers of adult females, adult males, and children in each of the ten most populous cities in each state:

```
Dim statesAnd10CitiesAndPeople(1 to 50, 1 to 10, 1 to 3) As Double
statesAnd10CitiesAndPeople(1,1,1) = 120748
' Number of adult males in Birmingham, Alabama.
statesAnd10CitiesAndPeople(1,1,2) = 145104
' Number of adult females in Birmingham, Alabama.
' ...
statesAnd10CitiesAndPeople(2,1,1) = 116381
' Number of adult males in Anchorage, Alaska.
statesAnd10CitiesAndPeople(2,1,2) = 109957
' Number of adult females in Anchorage, Alaska.
'...
Print StatesAnd10CitiesAndPeople(1,1,2)
' Output: 145104
```

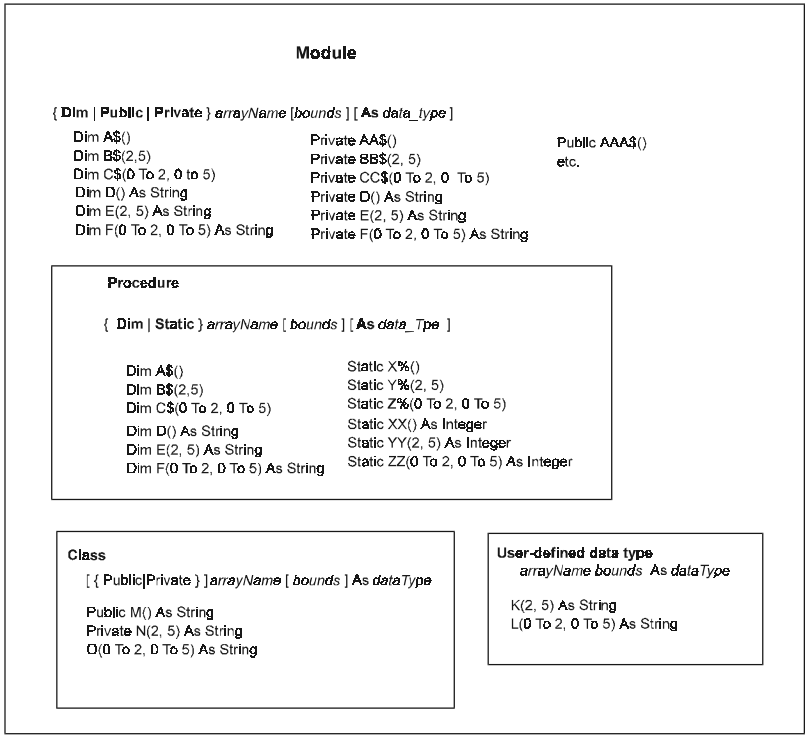
A LotusScript array can have as many as eight dimensions.

The size of an array—the number of dimensions and the extent of each individual dimension—is defined by the array's **bounds list**. Each dimension has a lower bound and an upper bound, specified as integer values.

LotusScript supports both **fixed** and **dynamic** arrays.

- You declare a fixed array once. At compile time, its size and storage requirements are set according to the specifications of its bounds list and the data type of its elements. At run time, storage is allocated for its elements, which are initialized like any ordinary variable of that data type. The array cannot be resized while the application is running.
- You declare a dynamic array once, but it can be sized and resized many times (with the ReDim statement) while the application is running. When you declare a dynamic array, you specify the data type of its future elements but include an empty bounds list, so LotusScript doesn't allocate space in memory for those elements. You resize a dynamic array at run time when you know how many elements you want it to be able to hold, at which time LotusScript allocates the necessary storage space. The values of the elements of the array can be reinitialized or preserved each time you resize the array.

You declare an array with the Dim statement or one of its variations, as summarized in the following diagram:



The syntactic elements in the declaration of an array are summarized in the following table:

Element	*Description*
Dim	Declares an array with Private scope.
Public, Private	Public declares an array with Public scope. Private declares an array with Private scope.
Static	Only applicable to arrays declared inside a procedure. Static arrays retain their values (rather than going out of existence) between calls to the procedure while the module remains loaded.
arrayName	The name of the array. At module level or within a procedure, *arrayName* can end in one or another of the data type suffixes that LotusScript recognizes. This determines the type of data that the array can hold. You can append a data type suffix character to the name of an array only if you do not include the As *dataType* clause in the declaration.
bounds	A comma-separated list of bounds for each dimension of *arrayName*. The bounds for each dimension are specified in the form: [*lowerBound* **To**] *upperBound* The *lowerBound* is the minimum subscript allowed for the dimension, and *upperBound* is the maximum. If no *lowerBound* is specified, the lower bound for the array dimension defaults to 0, unless the default lower bound has been changed to 1 using the Option Base statement. Array subscript bounds must fall in the range -32768 to 32767 inclusive. For a fixed array, bounds must be integer constants, that is, values known at compile time.
As *dataType*	Specifies the type of data the array can hold. Required in the declaration of an array within the definition of a user-defined data type or class, but optional in the declaration of a variable at module level or within a procedure. If you include this clause, *arrayName* cannot end in a data type suffix character. *dataType* can be any of the scalar data types, Variant, a user-defined data type, or an object reference.

Fixed arrays

Typically, you use a fixed array to manipulate a set of elements whose number is known at compile time and not subject to change while the application is running. For example, you might use a fixed array to match the names of employees with parking spaces in the company's garage by floor, section, and space number, since the number of floors, sections, and spaces is constant. A description of how you might do this will illustrate how to declare a fixed array, assign values to its elements, and refer to those values once they have been assigned.

Suppose that the garage has three floors, each floor is divided into four equal sections, and each section holds ten parking spaces. Here are two ways in which you might organize the information about these 120 parking spaces and the employees assigned to them:

```
' The first way uses a two-dimensional array.
' The array contains 480 elements, representing
' 4 pieces of information about each of 120
' parking spaces. When you refer to a given element
' in this array by its two subscripts, the first
' subscript identifies the parking space, and the
' second subscript identifies its floor, section,
' space number, or the person assigned to it.
Dim empSpacesA(1 To 120, 1 To 4) As String
empSpacesA(1,1) = "Floor 1"
empSpacesA(1,2) = "Section 1"
empSpacesA(1,3) = "Space 1"
empSpacesA(1,4) = "Maria Jones"
empSpacesA(2,1) = "Floor 1"
empSpacesA(2,2) = "Section 1"
empSpacesA(2,3) = "Space 2"
empSpacesA(2,4) = "Fred Smith"
' And so on down to the last space.
empSpacesA(120,1) = "Floor 3"
empSpacesA(120,2) = "Section 4"
empSpacesA(120,3) = "Space 10"
empSpacesA(120,4) = "Sal Piccio"
' Print information about Fred Smith's space.
Print empSpacesA(2,1) & " " & empSpacesA(2,2) & " " _
   empSpacesA(2,3) & " " empSpacesA(2,4)
' Output: Floor 1 Section 1 Space 2 Fred Smith

' The second way uses a three-dimensional array.
' The array contains 120 elements, each holding
' the name of the person assigned to a parking space.
' The three subscripts that identify a given element
' in this array correspond to the floor, section, and
' space to which that person has been assigned.
Dim empSpacesB(1 To 3, 1 To 4, 1 To 10) As String
empSpacesB(1,1,1) = "Maria Jones"
empSpacesB(1,1,2) = "Fred Smith"
' And so on down to the last space.
empSpacesB(3,4,10) = "Sal Piccio"
' Print information about Fred Smith's space.
Print "Floor 1 Section 1 Space 2 " & empSpacesB(1,1,2)
' Output: Floor 1 Section 1 Space 2 Fred Smith
```

Each of these two approaches involves declaring a multidimensional fixed array whose elements are of type String. While each array contains the same amount of information about each parking space, they have a different number of dimensions and elements, and they require you to use somewhat different strategies for entering and retrieving the information about each parking space.

Declaring a fixed array

When you declare a fixed array, you specify the data type, the number, and the organization of the elements that it will hold. You specify the data type of an array's elements in the As *dataType* clause of the declaration:

```
' Declare a one-dimensional array of strings.
Dim aStringArray(1 To 10) As String
' Declare a two-dimensional array of Variants.
Dim myVarArrayV(1 To 10, 1 To 10) As Variant
```

If the values that the array is going to hold are of one of the scalar data types that LotusScript recognizes, you can omit the As *dataType* clause and instead specify the data type by appending the appropriate data type suffix character to the name of the array:

```
' Declare a one-dimensional array of strings.
Dim aStringArray$(1 To 10)
' Declare a two-dimensional array of integers.
Dim anIntArray%(1 To 10, 1 To 10)
```

If you omit both the suffix character and the As *dataType* clause, LotusScript checks to see if the array name is covered by any applicable Def*type* statement. If it is, LotusScript defines the array's elements to be of the appropriate data type. Otherwise, LotusScript defines them to be of type Variant:

```
DefInt A-C
' Declare an array of integers.
Dim arrayOfInts(1 To 10)
' Declare an array of Variants.
Dim otherArrayV(1 To 10)
```

You specify the number of elements in an array and the number of dimensions along which they are organized in the bounds list. The lower and upper bounds of an array dimension can be any numeric constant between -32768 and 32767, inclusive, though the constraint that an array can take up no more than 64K of storage means that the range between lower and upper bounds in a multidimensional array must be smaller than this. The memory needed for an array depends on the size of the array and the storage needed for an element of the array. The size of an array is the total size of the elements in it. It is the product of the sizes of all the dimensions. For example:

```
Dim arrayOfSingles(1 To 5, 1 To 10, 1 To 2) As Single
```

The dimensional lengths are 5, 10, and 2, so arrayOfSingles holds 100 elements. The actual storage needed for all of these elements is 400 bytes, since one value of Single data type takes up four bytes of storage. Similarly,

```
Dim myStats(1980 To 1983, 1 To 4, -2 To 2) As Currency
```

Here the dimensional lengths are 4, 4, and 5 (1980, 1981, 1982, 1983; 1, 2, 3, 4; -2, -1, 0, 1, 2) for a total of 80 elements, each of which requires 8 bytes of storage. The amount of memory necessary to store myStats is therefore 640 bytes.

You might use such an array like myStats to hold some number of values distributed over a bell curve for each quarter of the years from 1980 to 1983 inclusive. The reason why you might use the subscript ranges 1980 To 1983, 1 To 4, and -2 To 2 instead of 1 To 4, 1 To 4, and 1 To 5 is to have a mnemonic device to make entering and retrieving values in the array more intuitive: to enter the value for the bottom of the curve in the second quarter of 1982, you would use a statement like this:

```
myStats(1982, 2, -2) = 123.456
```

This example demonstrates that a dimension's lower bound doesn't have to be 1, and that there are some cases where it is useful to have it be some other value. More often, however, it is convenient to have a dimension's lower bound be 1 or 0. LotusScript lets you set 1 or 0 as the default lower bound for the dimensions of all arrays that you declare in a module. You do this by including the appropriate Option Base statement in the module. If you then omit the lower bound subscript for a dimension when you declare an array, LotusScript automatically assigns it the appropriate value. Option Base 0 is the LotusScript language default but your product may choose a different setting, which you can override. For example:

```
Option Base 0
' Declare a 120 x 4 array both of whose dimensions
' are zero origin. This is the same as saying
' Dim empSpacesA(0 To 119, 0 To 3) As String
Dim empSpacesA(119, 3) As String

' Declare a 3 x 4 x 10 array all of whose dimensions
' are zero origin. This is the same as saying
' Dim EmpSpacesB(0 To 2, 0 To 3, 0 To 9) As String
Dim empSpacesB(2, 3, 9) As String
```

Or:

```
Option Base 1
' Declare a 120 x 4 array both of whose dimensions
' are one origin. This is the same as saying
' Dim empSpacesA(1 To 120, 1 To 4) As String
Dim empSpacesA(120, 4) As String

' Declare a 3 x 4 x 10 array all of whose dimensions
' are one origin. This is the same as
' Dim EmpSpacesB(1 To 3, 1 To 4, 1 To 10) As String
Dim empSpacesB(3, 4, 10) As String
```

You can mix and match explicit and implicit lower bound specifications in a declaration:

```
Option Base 0
Dim myStats(3, 1 To 2, -2 To 2) As Currency
' The first dimension of this 4 x 2 x 5 array is 0 To 3.

Dim arrayOfSingles(1 To 5, 9, 1) As Single
' The second and third dimensions of this 5 x 10 x 2 array
' are 0 To 9 and 0 To 1, respectively.
```

Use the LBound function to ascertain the lower bound of a dimension. The syntax is:

LBound (*arrayName* [, *dimension*] **)**

where *arrayName* is the name of the array, and *dimension* is an integer that represents the dimension whose lower bound you want to ascertain. The default value of *dimension* is 1. So, for example:

```
Option Base 1
Dim myStats(1980 To 1983, 2, -2 To 2) As Currency
Print LBound(myStats)
' Output: 1980 (the lower bound of the first dimension).
Print LBound(myStats, 2)
' Output: 1 (the lower bound of the second dimension).
```

You can ascertain the upper bound of a dimension with the UBound function.

Referring to the elements of an array

How you assign or refer to values in an array depends on the data type of the array's elements. This section describes how to assign values and refer to array elements of one or another of the scalar data types. For a description of how to do do this when an array holds object references or values of a user-defined data type, see Chapter 5 ("Creating User-Defined Data Types and Classes").

You assign a scalar value to an element in an array with a statement of the following form:

arrayName(*S1*, *S2*, *S3*,...) = *value*

where *arrayName* is the name of the array; *S1*, *S2*, *S3*,... are subscripts, one for each dimension of the array; and *value* is the value you want to assign to the element whose location in the array is defined by *S1*, *S2*, *S3*,... For example:

```
Option Base 1
Dim empSpacesB(3,4,10) As String
empSpacesB(1,1, ) = "Maria Jones"
empSpacesB(1,1,2) = "Fred Smith"
```

Or:

```
Dim empSpacesA(120,4) As String
Dim counter As Integer
Dim LB1 As Integer
Dim LB2 As Integer
' Get lower bound of first dimension.
LB1% = LBound(empSpacesA, 1)
' Get lower bound of second dimension.
LB2% = LBound(empSpacesA, 2)
' For the first 40 elements in the first dimension,
' assign the value "Floor 1" to the first element
' in the second dimension; for the next 40 elements
' in the first dimension, assign the value "Floor 2"
' to the first element in the second dimension; and
' for the last 40, assign the value "Floor 3".
For counter% = LB1% to LB1% + 40
     empSpacesA(counter%, LB2%) = "Floor 1"
     empSpacesA(counter% + 40, LB2%) = "Floor 2"
     empSpacesA(counter% + 80, LB2%) = "Floor 3"
Next
```

You refer to the value of a scalar element in an array by the element's subscripts, as in the following example which searches for parking spaces to which no employee has been assigned:

```
Option Base 1
Dim empSpacesB(3,4,10) As String
' Declare three String variables the quickest way
' to hold values for floor, section, and space.
Dim Flo$, Sec$, Spa$
' Declare six Integer variables the quickest way
' to hold values for the lower and upper bounds
' of the dimensions of empSpacesB for easy reference.
Dim LB1%, LB2%, LB3%, UB1%, UB2%, UB3%
```

```
' Initialize the array. Typically you do this by reading
' the data from a file rather than by hard-coding the
' values.
empSpacesB(1,1,1) = "Maria Jones"
empSpacesB(1,1,2) = ""
empSpacesB(1,1,3) = "Joe Smith"
' And so on down to the last space.
empSpacesB(3,4,10) = "Sal Piccio"

' Assign the lower and upper bounds of each dimension
' of empSpacesB to a variable.
LB1% = LBound(empSpacesB, 1)
LB2% = LBound(empSpacesB, 2)
LB3% = LBound(empSpacesB, 3)
UB1% = UBound(empSpacesB, 1)
UB2% = UBound(empSpacesB, 2)
UB3% = UBound(empSpacesB, 3)

' Loop through all the array elements and print
' the floor, section, and location of each space
' that has the empty string—that is, no employee name—
' as its value. Convert the floor, section, and space
' numbers to strings by calling the cStr function and
' passing it the appropriate subscript.
For counter1% = LB1% to UB1%
  For counter2% = LB2% to UB2%
    For counter3% = LB3% to UB3%
      If empSpacesB(counter1%, counter2%, counter3%) = "" Then
         Flo$ = "Floor " & cStr(counter1%) & " "
         Sec$ = "Section " & cStr(counter2%) & " "
         Spa$ = "Space " & cStr(counter3%) & " "
         Print Flo$ & Sec$ & Spa$ & "is empty."
      End If
    Next
  Next
Next
```

Dynamic arrays

You use a dynamic array if you want to defer declaring the number of the array's elements and dimensions until run time, or if you want to vary the array size at one or more points during execution of the application. To declare a dynamic array, you use a Dim statement (or one of its variations) with an empty subscript list (empty parentheses), as in the following example:

```
Dim myDynamicArray() As String
```

Since this Dim statement contains no information about the array's dimensions, the statement simply reserves the name myDynamicArray as the name of a dynamic array whose elements will be of type String: when you declare a dynamic array, it has no dimensions or elements, and no storage is allocated for it. The array is unusable until you specify its dimensions and their bounds in a ReDim statement, which defines the array type and size, and allocates storage for the elements and initializes them. The syntax of the ReDim statement is:

ReDim [**Preserve**] *arrayName* **(** *bounds* **)** [**As** *dataType*]

where *arrayName* is the name of an array that you previously declared with an empty bounds list, *bounds* is the bounds list with which you now want to define the number and extent of the array's dimensions, and As *dataType* specifies the data type of the elements that the array will hold. This must be the same as the data type in the original Dim statement. The optional Preserve keyword instructs LotusScript to retain the current values of the elements in *arrayName*. This is useful if you have declared a dynamic array with Dim, defined its size with ReDim, assigned values to its elements, and then want to expand the array to accommodate additional elements and assign them values, as in the following example:

```
Option Base 1
' Declare a dynamic String array. Later, this is
' defined as a one-dimensional array whose elements
' are assigned values that the user enters.
Dim myNames() As String
Dim ans1 As Integer
Dim ans2 As Integer
Dim counter As Integer
Dim userInput As String
' Ask the user to enter a number and assign it to ans1%.
ans1% = CInt(InputBox$("How many names would you like to enter?"))
' Use ans1% as the upper bound of the array's only dimension.
ReDim myNames(ans1%)
' Elicit ans1% strings from the user, and assign them
' to successive elements in the array.
For counter% = 1 to ans1%
  myNames(counter%) = InputBox$("Enter a name: ")
Next
' Print the contents of the array on a single line
' with a space between the value of each element.
For counter% = 1 to ans1%
   Print myNames(counter%) " " ;
Next
' Output: a newline
Print ""
```

```
' Ask the user for another number and assign it to ans2%.
ans2% = CInt(InputBox$("How many more names?"))
' If the number is greater than 0, resize the
' array, preserving its original values, so that the
' user can enter additional values.
If ans2% > 0 Then
  ReDim Preserve myNames(ans1% + ans2%)
  ' Elicit the new values and assign them to the
  ' elements that have been allocated after the old ones.
  For counter% = 1 to ans2%
    myNames(counter% + ans1%) = InputBox$("Enter a name: ")
  Next
  ' Print the contents of the array on a single line
  ' with a space between the value of each element.
   For counter% = 1 to ans1% + ans2%
    Print myNames(counter%) " " ;
  Next
  Print ""
End If
```

When you define the size of a dynamic array in the first ReDim statement that applies to it, this permanently defines the number of dimensions for that array. At any later time, you can change the values of any of the lower or upper bounds in the bounds list as long as the ReDim statement you use to do so does not include the Preserve keyword. When LotusScript encounters a ReDim statement that does not include the Preserve keyword, it reallocates the amount of storage for the array that the bounds list specifies and initializes the array's elements to the default values appropriate to their data type. If you do include Preserve in a ReDim statement, the only bound that LotusScript lets you change (by incrementing) is the upper bound of the last array dimension, in which case LotusScript allocates the appropriate amount of additional storage and initializes the additional array elements. You cannot change the number of dimensions of an array or the data type of its elements with a ReDim statement.

You can use the Erase statement to recover all of the storage currently allocated to a dynamic array. Applied to a fixed array, the Erase statement only reinitializes the array elements (to zeros, empty strings, EMPTY, or NOTHING, depending on the data type of the array's elements).

You can determine whether an identifier is the name of an existing array with the IsArray function. You can determine whether an array is a fixed array or a dynamic array with the DataType function, and you can ascertain the data type of an array's

elements with either the DataType or the TypeName function. You can use any of the LotusScript built-in functions that operate on scalar values to operate on the elements of an array, as in the following example:

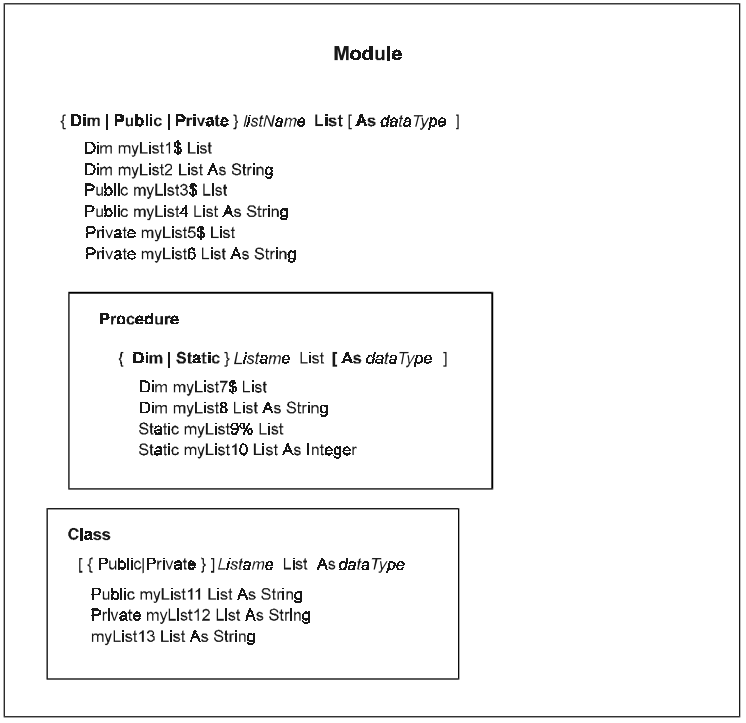
```
Dim myDblArray(1 To 10) As Double
Dim anIntArray(1 To 10) As Integer
Dim counter As Integer

' Seed the random number generator.
Randomize
' Populate myDblArray with random numbers
' greater than 0 and less than 1.
For counter% = 1 To 10
    myDblArray(counter%) = Rnd()
Next

' Populate anIntArray with the elements of myDblArray
' after rounding to one decimal place, multiplying
' by 10, dividing by 10 and adding 1 to the remainder
' to yield a whole number between 1 and 10.
For counter% = 1 To 10
    anIntArray(counter%) = _
      ((Round(myDblArray(counter%), 1) * 10) Mod 10) + 1
Next

' Test the first element of anIntArray for its data type.
Print TypeName(anIntArray(1))
' Output: INTEGER

' Print the contents of myDblArray and anIntArray.
For counter% = 1 To 10
   print myDblArray(counter%) & "    " & anIntArray(counter%)
Next
' Output: something like the following:
' .402520149946213    5
' .530154049396515    6
' .309299051761627    4
' 5.76847903430462E-02    2
' 2.41877790540457E-02    1
' .988802134990692    1
' .688120067119598    8
' .493557035923004    6
' .28598952293396    4
' .610387742519379    7
```

```
Dim aStringArray(1 to 5, 1 to 2)
aStringArray(1,1) = "Roman"
aStringArray(1,2) = "Minsky"
aStringArray(2,1) = "Sara"
aStringArray(2,2) = "Nala"
aStringArray(3,1) = "Raymond"
aStringArray(3,2) = "Nala"
aStringArray(4,1) = "Sandra"
aStringArray(4,2) = "Brooks"
aStringArray(5,1) = "Simon"
aStringArray(5,2) = "Anders"
' Check to see if the first two characters of each element
' in the first dimension of aStringArray would be SA
' if they were uppercase. If so, print the corresponding
' element in the second dimension of the array, making
' its first character uppercase and the rest lowercase.
For counter% = 1 to 5
   If UCase$(Left$(aStringArray(counter%, 1), 2)) = "SA" Then
      Print UCase$(Left$(aStringArray(counter%, 2), 1)) _
        & LCase$(Mid$(aStringArray(counter%, 2),  2, _
        Len(aStringArray(counter%, 2))))
  End If
Next
' Output:
' Nala
' Brooks
```

Lists

A **list** is a one-dimensional collection of elements of the same data type. In certain respects, a list resembles a one-dimensional dynamic array. LotusScript does not allocate any storage space at compile time for the elements of a list, and you can change the size of a list at any time while the application is running. Lists differ from dynamic arrays in being restricted to a single dimension and in their ability to automatically shrink or grow when elements are deleted from or added to them. A further difference is that you access each element in a list by a unique String value, called a **list tag**, rather than by a numeric subscript.

You can declare a list anywhere that you can declare a dynamic array—at module level, in a procedure, or in the definition of a class (but not in the definition of a user-defined data type). You declare a list with the Dim statement or one of its variations:



LotusScript follows the same conventions in assigning a data type to the elements of a list as it does to the elements of an array if you omit the As *dataType* clause from the Dim statement and do not include a data type suffix character in the list's name. If the name of the list is covered by a Def*type* statement, then LotusScript assigns that data type to the list's elements; otherwise, LotusScript makes them type Variant.

A list is initially empty. You add elements to it with statements of the following form:

*listName***(** *listTag* **)** = *value*

where *listName* is the name of the list, *listTag* is a string that uniquely identifies the element, and *value* is the value you want to assign to the element.

List tags are **case-sensitive** or **case-insensitive**, depending on the setting for case sensitivity in the module in which the list is declared. If case sensitivity is in effect for the module, the list tags "A123" and "a123" are considered to be different tags; whereas if case sensitivity is not in effect, they are considered to be the same and can be used interchangeably. You can control whether case sensitivity is observed in string comparison in a module by including the Option Compare statement in that module. The syntax is

Option Compare { **Case** | **NoCase | Binary** }

If you include the Case or Binary keyword, string comparison is case-sensitive in the module. NoCase means that such comparisons are case-insensitive. Option Compare Case is the default.

The following example illustrates how to declare a list, add elements to it, and refer to those elements. The elements in the list are of one of the scalar data types (String). See Chapter 5, "Creating User-Defined Data Types and Classes," for examples of lists containing object references and values of a user-defined data type.

```
' Make string comparison case-insensitive
' in this module.
Option Compare NoCase
' Declare a list—myList—to hold first names.
' The list tags will be unique IDs.
Dim myList List As String
Dim newTag As String
Dim newValue As String
' Put some elements in the list.
myList("A1234") = "Andrea"
myList("A2345") = "Vera"
myList("A3456") = "Isabel"
' Ask the user to enter an ID and a name.
newTag$ = InputBox$("Please enter your ID:")
newValue$ = InputBox$("Please enter your first name:")
' Add a new element to the list with
' the user's ID as the list tag and the user's name as
' the value of the new element.
myList(newTag$) = newValue$
Print myList(newTag$)
' Output: the name that the user entered
```

Working with lists

LotusScript provides a number of functions and statements for use with lists. These include TypeName, DataType, IsList, IsElement, ListTag, ForAll, and Erase. You use TypeName and DataType to ascertain the data type of a list's elements. You use IsList to determine whether an identifier is the name of an existing list, and you use IsElement to determine whether an identifier is the list tag for an element in a particular list.

TypeName(*listName*) returns a string of the form *dataType* LIST, for example, STRING LIST, where *dataType* is the data type that appeared or was implicit in the statement that declared the list.

TypeName(*listName*(*listTag*)) returns a string of the form *dataType*, for example, STRING, where *dataType* is the data type of the specified list element. You might test for the data type of an individual element in a list when the list has been declared to be of type Variant, since Variants can hold data of a variety of types.

DataType(*listName*) returns an integer equal to 2048 + *dataTypeCode*, for example, 2056 (2048 + 8, that is, the code for List + the code for String).

DataType(*listName*(*listTag*)) returns an integer representing the data type code of the specified element, for example, 8 (the code for String).

IsList(*listName*) returns True (-1) or False (0) depending on whether *listName* is a list.

IsElement(*listName* (*stringExpr*)) returns True (-1) or False (0) depending on whether *stringExpr* is a list tag in *listName*. There are a variety of circumstances under which you might want to test for the existence of a particular list tag in a list. Two cases are:

- You want to add a new element to a list and want to make sure that the list tag you plan to use isn't already in use (because if it is, and you used it in an assignment statement, you would overwrite the element that it identifies).
- You want to refer to an element and want to make sure that the element exists before doing so (because if you refer to a nonexistent list tag, LotusScript returns an error).

ListTag(*refVar*) returns the list tag of the element currently being processed in a ForAll loop**.** The *refVar* argument is the reference variable in a ForAll loop.

LotusScript executes the statements in a ForAll *refVar* In *container* block for each element in the list identified by *container* (for a detailed description of ForAll, see Chapter 7, "Directing Traffic Within an Application").

Erase *listName* removes all the elements in *listName* and reclaims the storage previously allocated to them. Erase *listName*(*listTag*) removes the individual element identified by *listTag* from the list and reclaims the storage previously allocated to it, leaving the rest of the list intact.

These functions are illustrated in the following example, which removes an employee's access to a parking space when the user enters a valid employee name (a valid list tag) and matching employee ID:

```
' Declare a list to hold employee IDs.
' The list tags will be the names of the employees.
Dim empList List As Double
' Make absolutely sure empList is Double.
If TypeName(empList) <> "DOUBLE LIST" Then
  Print "Warning: empList is " & TypeName(empList)
End If
If DataType(empList) <> 2053 Then
  Print "Warning: empList is " & CStr(DataType(empList))
  ' We expected 2053 (that is, 2048 + 5).
End If
' Declare a String variable for user name.
Dim ans As String
' Declare a Double variable for user ID.
Dim yourID As Double
' Declare an Integer variable to serve as a flag.
Dim found As Integer
' Create some list elements and assign them values.
empList("Maria Jones") = 12345
empList("Roman Minsky") = 23456
empList("Joe Smith") = 34567
empList("Sal Piccio") = 91234
' Ask the user to enter the name to be removed from the
' list of employees who have been assigned parking spaces.
ans$ = InputBox$("Which employee no longer needs a space?")
' Check to see if the employee's name appears as a list tag
' in the list. If not, display a message and stop. Otherwise,
' validate the employee's ID. If everything checks out,
' remove the employee item from the parking list.
If IsElement(empList(ans$)) = True then
    Print ans$ & "  is a valid employee name."
    yourID# = CDbl(InputBox$("What's " & ans$ & "'s ID?"))
    ' The following ForAll block does two things:
    ' it checks to see if yourID# is a valid ID and,
    ' if so, if it matches the ID for the employee
```

```
    ' whose name is ans$. If so, that element is removed
    ' (erased) from the list. The found% flag is initially
    ' FALSE (0). If yourID# is a valid ID, found% is set to
    ' TRUE (-1). The variable empID is the reference variable
    'in the ForAll loop.
    found% = FALSE
    ForAll empID In empList
        If empID = yourID# then
           found% = TRUE
           If ListTag(empID) = ans$ then
              Erase empList(ans$)
              ' Verify the removal of the list element.
              If  IsElement(empList(ans$)) = FALSE then
                Print ans$ & " is no longer on the list."
             End If
           Else
              Print "Valid ID but wrong employee."
           End If
           ' No need to look farther for yourID#,
           ' so get out of the ForAll loop.
           Exit ForAll
        End If
     End ForAll
     If found% = False then
        Print "No such employee ID."
     End If
Else
   Print "No such employee."
End if
```

Variants

Variant is a special data type: variables of type Variant can hold values of any of the data types that LotusScript recognizes, except for user-defined data types. The range of operations you can perform on these variables is remarkably broad. A variable of type Variant can hold any of the following:

- A value of any of the scalar data types that LotusScript supports—Integer, Short, Long, Double, Currency, String
- A Boolean value (see “Boolean values” later in this chapter)
- A date/time value (see “Dates” later in this chapter)
- An array or list

- An object reference, that is, a pointer to an OLE Automation object or to an instance of a product-defined or user-defined class
- The NULL value
- The EMPTY value

You declare a Variant variable the same way you declare a scalar variable—explicitly or implicitly. By default, a variable that you declare without using an As *dataType* clause or a data type suffix character character is of type Variant if no Def*type* statements are applicable. In this book, Variant variables appear with the suffix character V to distinguish them from object reference variables or variables of some user-defined data type. For example:

```
Dim myVariant1V As Variant
Dim myVariant2V
Public myVariant3V As Variant
myVariant4V = 123.45
```

When you declare a Variant variable explicitly, LotusScript initializes it to the special value EMPTY. (Use the function IsEmpty to test a Variant variable for this value.) When you assign a Variant variable a value, LotusScript determines the data type of that value in either of two ways, depending on the available information:

- If the data type of the value is known, then the value retains its original data type.
- If the value is a literal, it is assigned a default data type appropriate to that value.

You can determine the data type of a value assigned to a Variant variable with the DataType or TypeName function, as in the following example:

```
Dim numVarV As Variant
Dim anAmount As Currency
anAmount@ = 20.05
numVarV = anAmount@
Print TypeName(numVarV)
' Output: CURRENCY
numVar = 20.05
Print TypeName(numVar)
' Output: DOUBLE
```

Under certain circumstances, the data type of a value assigned to a Variant variable can change to accommodate the requirements of a particular operation on it. For instance, in the following example the user enters a sequence of numeric characters, which are then treated as a String value for some operations and as a numeric value for others:

```
' Declare an Integer variable and assign it an initial
' value of FALSE (0). The application subsequently tests
' this variable, taking appropriate action depending on the
' variable's value—True (-1) or False (0).
quitFlag% = FALSE
Dim ansV As Variant
' Have the user enter some numeric characters.
ansV = InputBox("Enter a number.")
' See how many characters the user entered
' and assign that number to the Integer variable
' UB%. This involves treating the value of ansV
' as a String.
UB% = Len(ansV)
' Test the value of ansV to see if it can be
' interpreted as being of one of the numeric
' data types. If so, declare a dynamic array of Variants,
' then allocate space for as many elements as
' there are characters in ansV, and then assign
' the successive digits in ansV to the elements in
' the array.
If IsNumeric(ansV) = True then
  Dim digitArrayV() As Variant
  ReDim digitArrayV(1 To UB%)As Variant
  For x% = 1 to UB%
     digitArrayV(x%) = Mid(ansV, x%, 1)
  Next
Else
   Print "You entered some nonnumeric characters."
   quitFlag% = TRUE
End If
```

```
' If ansV was able to be interpreted as a numeric,
' print its digits and their sum; then print
' the result of adding that sum to the original
' number that the user entered.
If quitFlag% = False Then
   Dim theSum As Integer
   ' theSum% is initialized to 0.
   For x% = 1 to UB%
      theSum% = theSum% + digitArrayV(x%)
      Print digitArrayV(x%) ;
   Next
   Print ""
   Print "Their sum is: " & theSum%
   Print "Their sum added to the original number is: " _
     & ansV + theSum%
End If
' Output, supposing the user enters 12345:
' 12345
' Their sum is: 15
' Their sum added to the original number is: 12360
```

Boolean values

LotusScript recognizes the Boolean values True and False, which it evaluates as -1 and 0, respectively. When you assign a Boolean value to a variable of type Variant, you can display that value as text ("True" or "False") or as an integer (-1 or 0), as the following example shows:

```
Dim varV As Variant
varV = 1 > 2          ' The expression 1 > 2 (1 is greater than 2)
                      ' evaluates to False, so varV is assigned a
                      ' value of False.
Print varV
' Output: False
Print TypeName(varV) ' Output: BOOLEAN
Print DataType(varV) ' Output: 11
varV = True
Print varV           ' Output: True
Print CInt(varV)     ' Output: -1
Print varV + 2       ' Output: 1
```

You can assign a Boolean value of True or False to a variable of any of the numeric data types that LotusScript recognizes. LotusScript converts that value to an integer (-1 or 0):

```
Dim anInt As Integer
varV = True
anInt% = varV
Print anInt%
' Output: 0
Print TypeName(anInt%)
' Output: INTEGER
```

LotusScript interprets the values -1 and 0 as True and False, respectively:

```
varV = -1
Print varV          ' Output : -1
If varV = True Then Print "varV is True." Else Print "varV is False."
' Output: varV is True.

anInt% = 0
If anInt% = True then Print "True" Else print "False"
' Output: False
```

You can define a constant as a Boolean value:

```
Const YES = True
Print YES
' Output: True
Print TypeName(YES)
' Output: BOOLEAN

Dim varV As Variant
varV = YES
Print varV
' Output: True

Dim anInt As Integer
anInt% = YES
print anInt%
' Output: -1
```

Dates

LotusScript does not have a date/time data type as such: you can't declare a variable and restrict the type of data that it can hold to date/time values. However, LotusScript does recognize dates internally and provides a set of functions for entering, retrieving, and manipulating date/time values, which are stored as eight-byte (double) floating-point values, in which the integer part represents a serial day counted from 1/1/100 AD, and the fractional part represents the time as a

fraction of a day, measured from midnight. The range of allowable values for a date is -65,7434 (January 1, 100 AD) to 2,958,465 (December 31, 9999)—0 is December 30, 1899. You use Variant variables to hold and manipulate date/time values, which you can produce by calling one or another of the following functions:

- Now, which returns the current date and time.
- Date, which returns the current date without the time.
- DateNumber, which returns the date corresponding to the year, month, and day that you pass it as integer arguments. This function is particularly useful when year, month, or day are run-time values assigned to variables.
- DateValue, which returns the date corresponding to the date you pass it as a string argument in any of the formats that LotusScript recognizes as valid ones for dates.
- CDat, which converts the value you pass it to a date or date/time value.

You can use the DataType or TypeName functions to determine if a Variant variable holds a date or date/time value. If it does, DataType returns a value of 7, and TypeName returns DATE.

The following examples illustrate the various ways you can derive date and date/time values, how you can assign them to Variant variables, and some of the operations you can then perform on them, such as calculating a time span or determining the day of the week on which a given date will fall.

Suppose that today is October 26, 1994, and the time is 7:49:23 AM and you declare the following variables:

```
Dim theInstantV As Variant
Dim theDateV As Variant
Dim theDateValV As Variant
Dim myDate As String
```

You can do the following sorts of date manipulation:

- Get the current date and time by calling the function Now and then assign the result to a Variant variable, the InstantV:

  ```
  theInstantV = Now
  Print theInstantV
  ' Output: 10/26/94 7:49:23 AM
  ```

- Print the integers corresponding to the day of the month and the hour of the day:

  ```
  Print Day(theInstantV) & " " & Hour(theInstantV)
  ' Output: 26 7
  ```

- Get the current date and assign it to a Variant variable, theDateV:

```
theDateV = Date
Print theDateV
' Output: 10/26/94
Print theDateV - 1
' Output: 10/25/94
```

- Convert the value of the current date to a value of type Double:

```
Print CDbl(theDateV)
' Output: 34633
' Convert a value of type Double
' to a date value, assign it to a
' Variant variable, and print it.
theDateV = CDat(34633)
Print theDateV
' Output: 10/26/94
```

- Get the integer representation of the current year, month, and day; increment the month and day values and assign the results to some Integer variables; pass them to DateNumber, which calculates the date on the basis of those values and returns it, assigning it to the Variant variable theDateV:

```
y% = Year(theDateV)
m% = Month(theDateV) + 1
d% = Day(theDateV) + 1
theDateV = DateNumber(y%, m%, d%)
Print theDateV
' Output: 11/27/94
```

- Assign a string that can be interpreted as a date to a String variable, myDate$; then convert it to a date/time value and perform a calculation on it (subtract a day) and return the resulting date:

```
myDate$ = "October 28, 1994"
Print DateValue(myDate$) - 1
' Output: 10/27/94
theDateV = DateValue(myDate$)
' Check the data type of the value
' held by the Variant variable theDateV.
Print TypeName(theDateV)
' Output: DATE
```

- Display the date in a particular print format:

```
Print Format(DateValue("10-18-14"), "mmm-d-yyyy")
' Output: Oct-18-1914
```

- Convert the date/time value of the current date to a value of type Double:

```
Print CDbl(Date)
' Output: 34633
```

- Convert the date/time value of a particular date to a value of type Double by passing it as a String to DateValue and then passing the result to CDbl, which converts it to a value of type Double:

```
Print CDbl(DateValue("10-18-14"))
' Output: 5405
Print CDbl(Date) - CDbl(DateValue("10-18-14"))
' Output: 29228
```

- Calculate the number of days between two dates:

```
theDateV = DateValue(Date)
theDateV = 10/26/94
y% = Year(theDateV)
m% = Month(theDateV) + 1
d% = Day(theDateV) + 1
theDateValV = DateNumber(y%, m%, d%)
' theDateValV = 11/27/94
Print CDbl(theDateValV) - CDbl(theDateV)
' Output: 32
```

- Determine which day of the week a particular day falls on—Sunday is 1.

```
Print Weekday(theDateValV)
' Output: 1
```

The following table summarizes the functions and statements that LotusScript provides for handling date/time values:

Function/Statement	*Purpose*
CDat Function	Converts a numeric or string expression to a date/time Variant value
Date Function	Returns the system date
Date Statement	Sets the system date
DateNumber Function	Converts year, month, and day, to a date value
DateValue Function	Converts a string to a date value
Day Function	Returns the day of the month (1-31) from a date/time expression
FileDateTime Function	Returns the date and time a file was most recently saved
Format Function	Formats a number, a date/time value, or a string
Hour Function	Returns the hour of the day (0-24) of a date/time expression
IsDate Function	Returns True (-1) if a Variant date/time value, otherwise False (0)
Minute Function	Returns the minute of the hour (0-60) from a date/time expression

Continued

Function/Statement	*Purpose*
Month Function	Returns the month of the year (1-12) from a date/time expression
Now Function	Returns the current system date and time
Second Function	Returns the current second of the minute (0-59) from a date/time expression
Time Function	Returns the system date
Time Statement	Sets the system date
TimeNumber Function	Converts hours, minutes, and seconds to a fractional date/time value
Timer Function	Returns the time elapsed since midnight in seconds
TimeValue Function	Converts a string to a fractional date/time value
Today Function	Returns the system date (equivalent to the Date function)
WeekDay Function	Returns the day of the week (1-7) from a date/time expression
Year Function	Returns the year as a four-digit integer from a date/time expression

Referring to Variants

As the preceding examples suggest, you assign values to Variant variables the same way you assign values to variables of the other data types, the only difference being that many of the rules governing the match of value to data type don't apply. That is, you can assign a Variant variable a value of any of the scalar data types where assigning a value of one scalar data type to a variable of another scalar data type would produce an error, as in the following example:

```
Dim myVariantV As Variant
Dim myVariantArrayV(1 to 5) As Variant
Dim aString As String
Dim anInt As Integer
myVariantV = 1234567
myVariantArrayV(1) = 1234567
myVariantV = "Hello"
myVariantArrayV(1) = myVariantV
aString$ = 1234567
' Produce an error, because 1234567 is not a String.
anInt% = 1234567
' Produce an error because 1234567 is too large
' to be treated as an Integer.
```

You refer to Variant variables in the same way you refer to variables of the data types they represent.

Variants: a footnote on usage

The Variant data type allows you a great deal of freedom in manipulating values of different types (including Booleans and dates) without having to concern yourself with type checking and compatibility issues. The Variant data type also makes it possible for arrays and lists to hold items of different data types (rather than being restricted to a single type) and significantly expands the range of data that you can include in a user-defined data type. However, Variants take up more storage than scalars, and operations involving Variants tend to be slower than those involving scalars. Furthermore, it is easy to lose track of the specific data type of a value that you are manipulating, which can sometimes produce unexpected results. It is therefore advisable to consider whether you really need to use a Variant variable, rather than a variable of one of the explicitly declared scalar types, to perform a given operation with efficiency.

Data Type Conversion

There are a variety of circumstances under which it is necessary to treat a value of one data type as though it were a value of a different data type or to perform an operation on a value of one data type to produce a value of another data type. These two related processes are referred to as **data type conversion.** Some form of data type conversion is involved when you add two numbers of different data types together, print the hexadecimal representation of a decimal number as a string, or calculate a date/time value (by treating that value as though it were a number). Sometimes you have to perform a data type conversion explicitly with one or another of the functions that LotusScript provides for that purpose, sometimes LotusScript performs the conversion automatically, and sometimes you can choose between the two methods of conversion. For example:

```
Dim aString As String
Dim aDouble As Double
Dim aFloat As Currency
Dim aVariantV As Variant

aString$ = "123.45"
aDouble# = 678.90

' Explicitly convert a string to a Currency value.
' That is, assign the return value of the conversion
' function CCur, which takes a String argument, to a variable
' of type Currency. Without type conversion, the statement
'   aFloat@ = aString$
' would produce an error.
aFloat@ = CCur(aString$)
Print aFloat@
' Output: 123.45
```

```
' Automatically convert a Double value
' to a Currency value by assignment. You
' could explicitly convert the value of
' aDouble# to a Currency value before
' assigning it to aFloat@. You might do
' this for the purposes of documentation.
aFloat@ = aDouble#
Print aFloat@
' Output: 678.9

' Automatically convert a Variant value
' of type String to a Currency value by
' addition, and then convert the
' resulting Currency value to a value
' of type Double by assignment. You can make
' both of these conversions explicit if you want.
aVariantV = aString$
aDouble# = aVariantV + aFloat@
Print aDouble#
' Output: 802.35
```

Explicit data type conversion

LotusScript provides several built-in functions for explicitly converting a value's data type. For a complete list and detailed description of these functions, see the *LotusScript Language Reference* or online Help. The following example includes a sampling of these functions and illustrates their use:

```
Dim aString As String
Dim anInt As Integer
Dim aDouble As Double
Dim myFixedPoint As Currency
Dim aVariantV as Variant

aString$ = "123"
' Convert the string "123" to a value of type Double.
aDouble# = CDbl(aString$)

' Convert the value of aDouble# to a string
' in hexadecimal notation.
aString$ = Hex$(aDouble#)
Print aString$
' Output: 7B

' Add the prefix &H to that string, to
' prepare the string for conversion to a
' hexadecimal number.
aString$ = "&H" & aString$
```

```
' Convert the string "&H7B" to an integer,
' add 12.46 to that integer, explicitly
' convert the result to a value of type Currency,
' and assign it to a variable of type Currency.
' If you omit the step of explicitly converting
' the integer to a value of type Currency, the
' conversion happens automatically when the
' assignment takes place.
myFixedPoint@ = CCur(CInt(aString$) + 12.46)
Print myFixedPoint@
' Output: 135.46

' Explicitly convert a value of type Currency
' to an integer, with automatic rounding off,
' and assign the result to a variable of type
' Integer. If you don't explicitly convert
' the Currency value to an integer,
' conversion (with rounding) happens
' automatically when the assignment takes place.
anInt% = CInt(myFixedPoint@) + 300
Print anInt%
' Output: 435

' Convert an integer to a date value
' and assign it to a Variant variable.
aVariantV = CDat(anInt%)
Print format$(aVariantV, "mm/dd/yyyy")
' Output: 03/10/1901

' Convert the date, format it, and assign the
' result to a String variable.
aString$ = Format$(aVariantV, "mmm-dd-yyyy")
Print aString$
' Output: Mar-10-1901
```

Automatic data type conversion

As mentioned earlier, LotusScript sometimes automatically converts values from one data type to another. This happens under the following circumstances:

- You assign a value of one numeric data type to a variable of a different numeric data type. In this case, LotusScript converts the data type of the value being assigned to the data type of the variable to which it is being assigned, if possible. For example: aDouble# = anInteger% assigns the value of the integer variable anInteger% to the double floating-point variable aDouble#, with the necessary conversion taking place automatically.

- You perform an arithmetic or comparison operation involving values of different numeric data types. When two numeric values with different data types are used as operands on either side of an arithmetic operator, LotusScript converts the data type of one operand to the data type of the other operand before the operation is evaluated, if possible. For example: aVariantV = anInteger% + aDouble# adds the values of anInteger% and aDouble#, treating them both as values of type Double. The result is then assigned to a Variant variable of type Double.

  When you compare two values of different numeric data types, LotusScript treats them as being of the same data type for the purpose of comparison. In the following example, the values of the variable anInt% and the variable myLong& are both treated as being of type Long:

  ```
  If anInt% > myLong& Then
    Print "The value of anInt% is greater than the value of myLong&."
  End If
  ```

- You increment the value of a Variant variable of some numeric type beyond the allowable limit for values of that type. For instance, in the following example, the statement aVariantV = aVariantV + 5 assigns a value of type Long, rather than a value of type Integer, to the Variant variable aVariantV because the largest value an Integer can have in LotusScript is 32767:

  ```
  aVariantV = 32767
  Print TypeName(aVariantV)     ' Output: INTEGER
  aVariantV = aVariantV + 5
  Print TypeName(aVariantV)     ' Output: LONG
  ```

- You add or concatenate the values of two Variant variables, one of which is of type String and the other of which is one of the numeric data types.

The following examples illustrate these various scenarios.

```
' This example illustrates the automatic conversion
' of decimal numbers to integers that happens when you perform
' integer division and when you assign a decimal number value
' to an integer variable.

Dim anInt As Integer
Dim aDouble As Double
' Do floating-point division.
anInt% = 12/7
Print anInt%
' Output: 2
aDouble# = 12/7
Print aDouble#
' Output: 1.71428571428571
```

```
' Do integer division.
anInt% = 12\7
Print anInt%
' Output: 1
aDouble# = 12\7
Print aDouble#
' Output: 1

' Do floating-point division.
anInt% = 12.5/2
Print anInt%
' Output: 6
aDouble# = 12.5/2
Print aDouble#
' Output: 6.25

' Do integer division.
anInt% = 12.5\2
Print anInt%
' Output: 6
aDouble# = 12.5\2
Print aDouble#
' Output: 6
```

In the next example, the value 1.6 is assigned to X. Since X is a variable of type Integer, 1.6 is converted to an integer before the assignment takes place. Conversion of floating-point values (Single and Double values) to integer values (Integer and Long values) rounds the value to the nearest integer, which is 2 in this case.

When 1.5 is assigned to Y, LotusScript rounds it to 2, the nearest even integer. A floating-point value exactly halfway between two integer values is always rounded to the nearest even integer value. So the value 2.5 is also rounded to 2 when it is assigned to Z. A value of 3.5 would be rounded to 4, a value of -3.5 would be rounded to -4, and so on. A value of .5 or -.5 is rounded to 0.

```
Dim X As Integer
Dim Y As Integer
Dim Z As Integer
X% = 1.6
Print X%
' Output: 2
Y% = 1.5
Print Y%
' Output: 2
Z% = 2.5
Print Z%
' Output: 2
```

The next example illustrates the way in which LotusScript handles data type conversion in Variant variables to accommodate numeric values. The general rule is that when a numeric value in a Variant variable exceeds the limit imposed by its data type, it is assigned the next data type in the following order: Integer, Long, Single, Double, Currency.

```
Dim sumV As Variant
Dim sInt As Integer
sInt% = 42
sumV = sInt%
Print TypeName(sumV)
' Output: INTEGER

' Assign the largest integer value to sInt%.
sInt% = 32767
sumV = sInt% + 1
' LotusScript converts sumV to a Long to prevent
' an overflow.
Print TypeName(SumV)
' Output: LONG
```

Finally, here's an example of how LotusScript does number-to-string and string-to-number conversion when a Variant variable is an operand in an operation involving the + operator, which can be used for both addition and string concatenation. Addition is performed when one of the following is true:

- Both operands contain numeric values.
- One operand is numeric, and the other is a Variant containing a string that can be interpreted as a number.
- Both operands are Variants, with a numeric value in one and a string value that can be interpreted as a number in the other.

Concatenation is performed when one of the following is true:

- Both operands are strings.
- One operand is a string that can't be interpreted as a number, and the other is a Variant containing a numeric value.

```
Dim aVariantV As Variant
aVariantV = 1040
Print TypeName(aVariantV)
' Output: INTEGER
```

```
Print aVariantV + "A"
' Output: 1040A
' because "A" is a string and 1040 can be interpreted as a string.
aVariantV = "43"
Print TypeName(aVariantV)
' Output: STRING
Print aVariantV + 5
' Output: 48
' because 48 is a number and 1040 can be interpreted as a number.
```

Chapter 4
Procedures: Functions, Subs, and Properties

Procedures (sometimes called **subprograms**) are discrete blocks of reusable code that eliminate redundancy in an application, making it easier to read, debug, and maintain. In a LotusScript application, a procedure can be:

- A LotusScript built-in function (such as, Abs or UCase)
- An @function in a Lotus product (for example, @Sum in Lotus 1-2-3®)
- A macro or agent in a Lotus product
- A C function in a Dynamic Link Library (DLL)
- A user-defined LotusScript function
- A user-defined LotusScript sub
- A user-defined LotusScript property

This chapter explains how you create and use functions, subs, and properties. You can create functions, subs, and properties in two areas of an application: at module level and as part of the definition of a user-defined class. This chapter focuses on functions, subs, and properties that you define at module level, while Chapter 5, "User-Defined Data Types and Classes," describes functions, subs, and properties that you define as components of a user-defined class. The same basic rules apply in defining and referring to functions, subs, and properties at module level and within a user-defined class. These are described in this chapter, which covers the following topics:

- How to declare, define, and execute a function
- How to declare, define, and execute a sub
- How to declare, define, and refer to a property

Functions

A **function** is a named procedure that returns a single value. LotusScript provides a set of built-in functions that you can use to perform a variety of common numeric, date/time, string, data-conversion, and value-testing operations, some of which are described in the preceding chapter and all of which are fully described in the *LotusScript Language Reference* and in online Help.

LotusScript also lets you create your own functions. You define a function by specifying a series of one or more statements that are to be executed as a block when the application calls the function. You enclose these statements between the function's **signature** and the End Function statement (which marks the end of the function's definition).

A function's signature specifies the function's name, its scope, the data types of the values that it expects the application to pass it (if any), the lifetime of the variables that it defines (if any), and the data type of the value it returns to the application.

The statements that comprise the body of a function can include the following:

- Variable declarations
- Assignment statements (including statements that assign values to the function itself)
- Calls to built-in functions
- Calls to user-defined procedures (including calls to the function itself)
- Looping and branching statements (including Exit Function and End, which cause execution of the function to terminate before reaching the block terminator)
- Statements for performing standard file operations and for communicating with the end user

Certain other statements and directives are not allowed within the body of a function, most notably, those that declare or define a function, sub, property, or user-defined data type or class, and the Option, Use, and UseLSX statements.

The rest of this section explains how to do the following:

- Declare a function
- Define a function
- Execute a function

Declaring and defining functions

When you declare a function, you make the information contained in the function's signature known to the application. When you define a function, you provide this information as well as the set of statements that are to be executed when the application calls the function.

Declaring a function allows you to refer to that function before you actually define it. Referring to an undeclared function before you define it can produce errors, as in the following example:

```
' Error-producing sequence
Function FirstFunction(yourName As String) As String
   ' This function calls SecondFunction.
      FirstFunction$ = SecondFunction$(yourName$)
End Function

Function SecondFunction(aString As String) As String
     SecondFunction$ = "Hi, " & aString$ & "."
End Function

Print FirstFunction$(InputBox$("What's your name?"))
' Suppose the user types Carol.
' Output: run-time error.
```

The reason that the Print statement produces an error here is that the LotusScript compiler makes a single pass over the code, interpreting the reference to the previously undeclared and undefined function SecondFunction$ in FirstFunction$ as a reference to an implicitly declared variable within the function definition. This reference produces a run-time error because LotusScript interprets SecondFunction$ as an undeclared array with an inappropriate subscript (in this case, the empty string).

There are two easy ways to avoid errors caused by referring to previously undefined functions:

- Define your functions, subs, and properties in the reverse order in which the application calls them. For example:

  ```
  ' Good sequence.
  Function SecondFunction(aString As String) As String
        SecondFunction$ = "Hi, " & aString$ & "."
  End Function
  Function FirstFunction(yourName As String) As String
        FirstFunction$ = SecondFunction$(yourName$)
  End Function

  Print FirstFunction$(InputBox$("What's your name?"))
  ' Suppose the user types Carol.
  ' Output: Hi, Carol
  ```

- Explicitly declare your functions, subs, and properties before you define them. You do this with Declare statements. This strategy is generally the easier and less error-prone. For example:

```
' Good sequence including Declare.
Declare Function FirstFunction(yourName As String) As String
Declare Function SecondFunction(aString As String) As String

Function FirstFunction(yourName As String) As String
      FirstFunction$ = SecondFunction$(yourName$)
End Function

Function SecondFunction(aString As String) As String
      SecondFunction$ = "Hi, " & aString$ & "."
End Function

Print FirstFunction$(InputBox$("What's your name?"))
' Suppose the user types Carol.
' Output: Hi, Carol
```

Declaring a function

You use the Declare statement to explicitly declare a function as a member of a user-defined class or at module level in a product that does not support the IDE. The IDE automatically generates a Declare statement for each function that you define at module level. You should therefore not include any Declare statements at module level if you are using the IDE. All Lotus products, regardless of the LotusScript application development environment, allow you to use Declare statements in the definition of a user-defined class. Declare statements should appear at the beginning of the class definition.

The syntax of the Declare statement is:

Declare [**Public** | **Private**] [**Static**] **Function** *functionName*

[**(** *parameterList* **)**] [**As** *dataType*]

The following table summarizes the elements of the Declare statement:

Element	*Description*
Public, Private	When you declare a function at module level, Public indicates that the application can refer to the function outside the module in which the function is defined, as long as that module is loaded. Private indicates that the function is available only within the module in which it is defined. When you declare a function inside the definition of a user-defined class, Public means that the function is available outside the class definition. Private means that the function is only available within the class definition. By default, functions defined at module level are Private, and functions defined within a class are Public.
Static	Declares variables defined within the function to be static by default. Static variables retain their values (rather than going out of existence) between calls to the function while the module in which it is defined remains loaded.
functionName	The name of the function, which can end in one or another of the LotusScript data type suffix characters (%, &, !, #, @, and $), which determine the data type of the function's return value. You can append a data type suffix character to a function name when you declare it only if you do not include the As *dataType* clause in the declaration.
parameterList	A comma-delimited list of the function's formal parameters (if any), enclosed in parentheses. (The list can be empty.) This list declares the variables for which the function expects to be passed values when it is called. Each member of the list has the following form: [**ByVal**] *paramName* [**()** | **List**] [**As** *dataType*] ByVal means that *paramName* is passed by value: that is, the value assigned to *paramName* is a local copy of a value in memory rather than a pointer to that value. *paramName()* is an array variable; List identifies *paramName* as a list variable; otherwise, *paramName* can be a variable of any of the other data types that LotusScript supports. You can't pass an array, a list, an object reference, or a user-defined data type structure by value. As *dataType* specifies the variable's data type. You can omit this clause and use a data type suffix character to declare the variable as one of the scalar data types. If you omit this clause and *paramName* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), its data type is Variant.
As *dataType*	Specifies the data type of the function's return value. A function can return a scalar value, a Variant, or an object reference. If you include this clause, *functionName* cannot end in a data type suffix character. If you omit this clause and *functionName* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), the function's return value is Variant.

Defining a function

You can define a function at module level or within the definition of a user-defined class. The syntax of the statement that defines a function is as follows:

[**Public** | **Private**] [**Static**] **Function** *functionName* [**(** *parameters* **)**] [**As** *dataType*]

statements

End Function

The table in the preceding section provides a synopsis of the elements of the Function statement.

Note that if you declare a function before defining it, the definition has to be essentially the same as the declaration. For example, the following declaration and definition are acceptable because Private is the default scope for a function defined at module level, and Cubit# is equivalent to Cubit As Double. Substituting Public for Private, adding Static, altering the contents of the argument list or changing any data types would produce an error:

```
Declare Private Function Cubit(intArg As Integer) As Double

Function Cubit#(intArg%)
  ' Calculate the cube of intArg% and
  ' make it the return value of Cubit#.
  Cubit# = intArg% ^ 3
End Function
```

Note You can include the appropriate data type suffix character in referring to a function whose return value is one of the scalar data types if you like, but you need not do so.

Values that a function can manipulate

Functions can manipulate values of the following sorts:

- Values contained in module-level variables that the function can access directly
- Values contained in member variables of a class that a function can access directly if it has been defined as a member of that class
- Values that the application passes to the function at run time either directly or by reference as **arguments** (sometimes called **actual parameters**) in the statement that calls the function
- Values contained in variables (known as **local variables**) that the function defines for its own use
- Values returned by another function that the function calls

The following sections describe the way a function handles module-level variables, the values that the application passes it as arguments when calling the function, and variables that a function defines for its own use. For information on functions defined as class members and how they handle member variables, see Chapter 5, "User-Defined Data Types and Classes."

Module-level variables

As long as a function doesn't define a local variable with the same name (see "Local Variables" below), it can access a variable defined at module level, as in the following example:

```
Dim anInt As Integer
Function ThreeTimes1 As Double
   ' Multiply the module-level variable anInt% by 3
   ' and assign the result as the function's return value.
   ThreeTimes1# = anInt% * 3
End Function
anInt% = 5
Print ThreeTimes1#
' Output: 15
```

Using procedures to directly manipulate module-level variables is not recommended. The practice goes against the grain of modular programming style and makes it easy to introduce errors into your application, especially if you don't always declare your variables explicitly.

Parameters

When you define a function, you can declare a list of variables (sometimes called **formal parameters** or, simply, **parameters**) as part of its signature. These variables are placeholders for values that the application passes the function at run time and that the function then uses when it executes. The run-time values that the application passes the function are known as **actual parameters** or **arguments**.

There are two ways in which the application can pass arguments to a function: **by value** and **by reference** (the default). Some types of argument—for example, an array—can only be passed by reference. When you pass an argument by value, you pass a copy of the value in memory. When you pass an argument by reference, you pass a pointer to the value in memory. This means that when a function changes the value of an argument that the caller passes it by value, the effect of the change is local to that function: the copy but not the original value in memory changes. However, when a function changes the value of an argument that the caller passes it by reference, the original value changes.

For example:

```
Dim A As Integer
Dim B As Integer
Function PlusFive(X As Integer, Y As Integer) As Integer
  ' Increment each of the two values received by 5
  ' and make their sum the function's return value.
  X% = X% + 5
  Y% = Y% + 5
  PlusFive% = X% + Y%
End Function
A% = 10
B% = 15
Print PlusFive%((A%), (B%))' Pass A% and B% by value.
' Output: 35
Print A% B%
' Output: 10  15           ' A% and B% are unchanged.
Print PlusFive(A%, B%)     ' Pass A% and B% by reference.
' Output: 35
Print A% B%                ' A% and B% are changed.
' Output: 15 20
```

Local variables

A procedure can define variables for its own use. By default, a local variable exists only as long as the procedure in which it is defined is executing. Then it ceases to refer to a location in memory, and its value is forgotten. If you include the Static keyword in the declaration of a local variable, that variable retains its address in memory, and its value persists between calls to the procedure. In either case, local variables are not visible outside of the procedure in which you define them though you can pass them as arguments to other procedures that the procedure calls.

You can define a local variable with the same name as a module-level variable. This is known as **shadowing**. When you do this, the procedure uses the local variable and ignores the module-level variable of the same name. For example, defining counter% as a local variable makes the BadCount example cited under "Module-level variables" earlier in this section work the way it should. The calling While loop executes three times, because BadCount no longer has any effect on the counter variable in the calling loop:

```
Dim counter As Integer          ' Module-level variable
Function BadCount As Integer
   Dim counter As Integer       ' Local variable
   counter% = 1
   While counter% < 4
     ' Do something.
     counter% = counter% +1
   Wend
   BadCount% = counter%
End Function
```

```
counter% = 1
While counter% < 4
  Print "BadCount% = " & BadCount%
  counter% = counter% +1
Wend
```

The following example illustrates the use of static and nonstatic local variables and shows how to pass a local variable as an argument in a call to another procedure. The example consists of two functions, GetID and FindMatch. GetId prompts the user for a password (his or her first name) and then calls FindMatch, passing it the password. FindMatch determines if the name is in the module-level array theNames. If it is, FindMatch returns a value of True (-1) and GetId displays a confirmation message. If the name is not in the array, FindMatch increments the static variable callCounter% by 1 and returns a value of False (0), at which point GetId displays a message box asking the user to try again or quit. If the user chooses to quit, GetId displays an appropriate message. If the user tries again, GetId again calls FindMatch to check the name. If the user enters three invalid names in a row (in three successive calls to FindMatch), FindMatch terminates the program.

```
%Include "LSCONST.LSS"
  ' Declare an array of Strings and initialize it with some names.
  Dim theNames(1 to 6) As String
  theNames(1) = "Alex"
  theNames(2) = "Leah"
  theNames(3) = "Monica"
  theNames(4) = "Martin"
  theNames(5) = "Elizabeth"
  theNames(6) = "Don"

Function FindMatch(yourName As String) As Integer
    Static callCounter As Integer ' To count the number of
                                  ' calls to FindMatch.
    Dim counter As Integer        ' Loop counter.
    FindMatch% = FALSE

   For counter% = 1 to 6
      If yourName$ = theNames(counter%) Then
        FindMatch% = TRUE
        Exit For                  ' Exit from the For loop now.
     End If
  Next

  ' If the user enters an invalid name,
  ' increment callCounter%.
  If FindMatch% = False Then callCounter% = callCounter% + 1
```

```
  ' After three consecutive invalid names, terminate the script.
   If callCounter% = 3 Then
      Print "Go away, friend."
      End                          ' Terminate execution.
   End If
End Function

Function GetId As String
    Dim match As Integer
    Dim goAgain As Integer
    Dim pswd As String
    Dim msg As String
    Dim msgSet As Integer
    Dim ans As Integer
    match% = FALSE
    goAgain% = TRUE
    msg$ = "That's not a valid name. Would you like to try again?"
    msgSet% = MB_YESNO + MB_ICONQUESTION

     ' Go through this While loop at least once.
     While match% = FALSE and goAgain% = TRUE
        pswd$ = InputBox$("Please enter your name.")
        ' Call FindMatch, passing it the name the user
        ' just entered (pswd$).
        match% = FindMatch%(pswd$)
        ' If the name the user entered isn't in theNames,
        ' see if the user would like to try again or quit.
        If match% = False Then
           ans% = MessageBox(msg$, msgSet%)
           ' If No, end the While loop.
             If ans% = IDNO Then
                goAgain% = FALSE
                GetID$ = "Have a nice day, " & pswd$ & "."
             End If
         Else
            GetID$ = "Your ID is valid, " & pswd$ & "."
         End If
     Wend
End Function
```

```
Print GetID$
' Output: (1) The user enters the name "Martin" at the prompt:
'              Your ID is valid, Martin.
' Output: (2) The user enters the name "Fred" at the prompt
'             and then selects No in the message box:
'              Have a nice day, Fred.
' Output: (3) The user enters the name "Fred" at the prompt,
'             then selects Yes, then enters "Frank," then selects
'             Yes, then enters "Joe":
'              Go away, friend.
```

Assigning a function a return value

Typically, one of the statements that you include in the definition of a function assigns the function a **return value**, that is, a value that it returns to the caller. This statement takes the form *FunctionName = returnValue,* where *returnValue* has the data type specified in the As *dataType* clause of the function's signature: a scalar, a Variant, or an object reference. For example,

```
Function Cubit(intArg%) As Double
  ' Return the cube of intArg%.
  Cubit# = intArg% ^ 3
End Function
```

or

```
Function Left5(aString As String) As String
  ' Return the leftmost 5 characters of aString$.
  Left5$ = Left$(aString$, 5)
End Function
```

You can cause a function to return an array or a list. To do so, you need to make the function's return value a Variant, which can hold an array or list, as in the following example, which passes an array of names in one format (first name, space, last name) to a function that returns another array in which the names appear in a different format (last name, comma, space, first name):

```
Dim myVariantVarV As Variant
Dim anArray(1 to 3) As String
Dim X As Integer
anArray$(1) = "Alex Smith"
anArray$(2) = "Elizabeth Jones"
anArray$(3) = "Martin Minsky"
```

```
Function SwitchNames(arrayOfNames() As String) As Variant
   ' Declare a local array variable to pass back to the
   ' application as the return value of SwitchNames. Performing
   ' the operation on arrayOfNames, which is passed by
   ' reference, would change anArray if arrayOfNames were
   ' the return value of the function.
   Dim newArrayOfNames(1 to 3) As String
   Dim tempArray(1 to 3, 1 to 3) as String
   Dim aSpace As Integer
   For X% = 1 to 3
     ' Locate the space that separates first name from last name
     ' in arrayOfNames, then extract everything before the
     ' space to tempArray, then extract everything after the
     ' space to the corresponding location in tempArray's
     ' second dimension.
     aSpace% = Instr(arrayOfNames$(X%), " ")
     tempArray$(1, X%) = Mid$(arrayOfNames$(X%), 1, aSpace% - 1)
     tempArray$(2, X%) = Mid$(arrayOfNames$(X%), aSpace% + 1, _
          Len(arrayOfNames$(X%)))
   Next
   For X% = 1 to 3
     newArrayOfNames(X%) = tempArray(2, X%) & ", " & tempArray(1, X%)
   Next
   SwitchNames = newArrayOfNames
End Function

MyVariantVarV = SwitchNames(anArray())
For X% = 1 to 3
   print myVariantVarV(x%)
Next
' Output: Smith, Alex
' Jones, Elizabeth
' Minsky, Martin
For x% = 1 to 3
   Print anArray(x%)
Next
' Output: Alex Smith
' Elizabeth Jones
' Martin Minsky
```

A function *need* not contain a statement that assigns it a return value. If you don't include such a statement when you define the function, LotusScript assigns the function the default return value appropriate to the data type specified or implied in

the function's signature. The default values are 0 for a numeric data type, the empty string ("") for a String, EMPTY for a Variant, and NOTHING for an object reference. For example:

```
Dim anInt As Integer
Dim anotherInt As Integer
Function PrintCube(intArg%) As Integer
    Print intArg% ^ 3
End Function
anInt% = CInt(InputBox$("Enter a number:"))
' Suppose the user enters 3.
anotherInt% = PrintCube%(anInt%)
' Output: 27
Print anotherInt%
' 0
```

Executing a user-defined function

There are various ways to execute a user-defined function. The possibilities differ according to the number of arguments that the function expects to be passed when you call it and whether the function appears as part of a statement (such as an assignment statement or a Print statement) or just by itself.

Executing a function that takes no arguments

When you call a parameterless function by including it in a statement, the function's name can end in empty parentheses or no parentheses, as you prefer. For example:

```
Dim anInt As Integer
Dim aDouble As Double
Function Cubit1 As Double
   ' Return the cube of anInt% and display a message
   ' saying what that value is.
   Cubit1# = anInt% ^ 3
   Print anInt% & " cubed = " & Cubit1# & "."
End Function
anInt% = 4
aDouble# = Cubit1#
' Output:  4 cubed is 64.
aDouble# = Cubit1#
' Output: 4 cubed is 64.
Print aDouble#
' Output: 64
Print Cubit1#
' Output: 4 cubed is 64.
          64
Print Cubit1#
' Output: 4 cubed is 64.
          64
```

You can call a parameterless function by simply entering the function's name, which must not include empty parentheses. For example:

```
Cubit1#
' Output: 4 cubed is 64
```

Executing a function that takes a single argument

When you call a function that expects a single argument, you must enclose that argument in parentheses when you include the function in a statement. For example:

```
Dim anInt As Integer
Dim aDouble As Double
Function Cubit2(X As Integer) As Double
   ' Return the cube of X% and display a message
   ' saying what that value is.
   Cubit2# = X% ^ 3
   Print X% & " cubed = " & Cubit2# & "."
End Function
anInt% = 4
aDouble# = Cubit2#(anInt%)
' Output: 4 cubed is 64.
Print aDouble#
' Output: 64
Print Cubit2#(anInt%)
' Output: 4 cubed is 64.
          64
```

You can call a one-parameter function in any of the following additional ways:

- With a Call statement. You must enclose the argument in parentheses.
- By entering the name of the function followed by the argument that it expects with no parentheses.
- By entering the name of the function followed by the argument it expects enclosed in parentheses. This notation signifies that you are passing the argument by value rather than by reference.

For example:

```
Call Cubit2#(anInt%)
' Output: 4 cubed is 64.
Cubit2# anInt%
' Output: 4 cubed is 64.  (anInt% is passed by reference.)
Cubit2#(anInt%)
' Output: 4 cubed is 64.  (anInt% is passed by value.)
```

Executing a function that takes multiple arguments

When you call a function that expects multiple arguments, you must enclose those arguments in parentheses when you include the function in a statement. For example:

```
Dim anotherInt As Integer
Function Cubit3(X As Integer, Y As Integer) As Double
    ' Return the product of X% and Y%.
    Cubit3# = X% * Y%
    Print X% & " times " Y% & " = " & Cubit3# & "."
End Function
anInt% = 4
anotherInt% = 6
Print Cubit3#(anInt%, anotherInt%)
' Output: 4 times 6 = 24.
          24
```

You can also call a function that expects multiple arguments with a Call statement or by entering the function's name followed by the arguments. The Call statement requires parentheses; the function name by itself does not allow parentheses. For example:

```
Call Cubit3#(anInt%, anotherInt%)
' Output: 4 times 6 = 24.
Cubit3# anInt%, anotherInt%
' Output: 4 times 6 = 24.
```

Executing a function recursively

A function can call itself. A function that calls itself is known as a **recursive function**. The following is a recursive function:

```
Function Facto# (theNum%)
   ' Calculate theNum% factorial and make it
   ' the return value of Facto#.
   If theNum% <= 0 Then
      Facto# = 0
   ElseIf theNum% = 1 Then
      Facto# = 1
   Else
      Facto# = theNum% * Facto#(theNum% -1)
   End If
End Function
```

All recursive functions can be rewritten as nonrecursive functions. Why write a recursive function instead of a nonrecursive one when the operation you want to perform lends itself to either treatment? The trade-off is elegance of code for inelegance of memory management: recursive functions tend to be more compact than their nonrecursive counterparts but they also tend to use a lot of memory.

Subs

A **sub** is a named procedure that performs one or more operations without returning a value to its caller. Except for not returning a value, a sub is not much different from a user-defined function: you define a sub by specifying a series of one or more statements that are to be executed as a block. You enclose these statements between the sub's signature and the End Sub statement, which marks the end of the sub's definition.

A sub's signature specifies the sub's name, its scope, the sorts of values that it expects the application to pass it (if any), and the lifetime of the variables that it defines (if any).

The statements that comprise the body of a sub can be any of the same kinds that comprise the body of a user-defined function, with one exception: you can't include a statement that assigns the sub a value. Statements that can't appear in the body of a function similarly can't appear in the body of a sub.

Essentially the same conventions and restrictions that apply in declaring and defining functions apply in declaring and defining subs: you can define a sub at module level or as a member of a user-defined class. Declaring a sub before you define it allows you to refer to that sub before you actually define it. The circumstances under which you should or should not use the Declare statement to declare a sub before referring to it are the same as those that apply in the case of user-defined functions (see "Declaring a function" earlier in this chapter).

The ways in which you can execute a user-defined sub are a subset of the ways in which you can execute a user-defined function. These are described in "Executing a sub" later in this chapter.

LotusScript recognizes four specialized kinds of sub that you can define—Sub Initialize, Sub Terminate, Sub New, and Sub Delete. These are subject to somewhat different rules than the ones for declaring, defining, and executing other user-defined subs. These are described separately in "Specialized subs" later in this chapter.

Declaring and defining subs

The syntax for declaring a sub is:

Declare [**Public | Private**] [**Static**] **Sub** *subName* [**(** *parameters* **)**]

The syntax for defining a sub is

[**Public | Private**] [**Static**] **Sub** *subName* [**(** *parameters* **)**]

statements

End Sub

The elements of these syntax diagrams are summarized in the following table:

Element	*Description*
Public, Private	When you declare a sub at module level, Public indicates that the application can refer to the sub outside the module in which it is defined, as long as that module is loaded. Private indicates that the sub is available only within the module in which it is defined. When you declare a sub inside the definition of a user-defined class, Public means that the sub is available outside the class definition. Private means that the sub is only available within the class definition. By default, subs defined at module level are Private, and subs defined within a class are Public.
Static	Declares variables defined within the sub to be static by default. Static variables retain their values (rather than going out of existence) between calls to the sub while the module in which it is defined remains loaded.
subName	The name of the sub.
parameterList	A comma-delimited list of the sub's formal parameters (if any), enclosed in parentheses. (The list can be empty.) This list declares the variables for which the sub expects to be passed values when it is called. Each member of the list has the following form: [**ByVal**] *paramName* [**()** | **List**] [**As** *dataType*] ByVal means that *paramName* is passed by value: that is, the value assigned to *paramName* is a local copy of a value in memory rather than a pointer to that value. *paramName()* is an array variable; List identifies *paramName* as a list variable; otherwise, *paramName* can be a variable of any of the other data types that LotusScript supports. You can't pass an array, a list, an object reference, or a user-defined data type structure by value. As *dataType* specifies the variable's data type. You can omit this clause and use a data type suffix character to declare the variable as one of the scalar data types. If you omit this clause and *paramName* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), its data type is Variant.

As with user-defined functions, the declaration and definition of a sub have to agree in their specifications (whether explicit or implicit) for the sub's scope, its name, and its parameter list.

Executing a sub

You can execute a user-defined sub in either of two ways: by including it in a Call statement or by simply entering its name followed by the arguments that it expects to be passed (if any). Calling conventions differ according to the number of arguments the sub expects to be passed and whether you use the Call statement to do the calling.

Executing a sub that takes no arguments

When you call a parameterless sub by including it in a Call statement, the sub's name can end in empty parentheses or no parentheses, as you prefer. For example:

```
Dim aName As String
Sub PrintName1
   ' Make the contents of firstName$ be all uppercase
   ' and display the result.
   firstName$ = UCase$(firstName$)
   Print firstName$
End Sub
firstName$ = "David"
Call PrintName1()
' Output: DAVID
Call PrintName1
' Output: DAVID
```

You can call a parameterless sub by simply entering the sub's name, which must not include empty parentheses. For example:

```
PrintName1
' Output: DAVID
```

Executing a sub that takes a single argument

When you call a sub that expects a single argument, you must enclose that argument in parentheses when you include it in a Call statement. Enclose the argument in single parentheses if you want to pass it by reference. To pass the argument by value, enclose it in double parentheses. For example:

```
Sub PrintName2(someName As String)
    ' Make the contents of someName$ be all uppercase
    ' and display the result. If someName$'s contents are
    ' passed by reference, change the value of the corresponding
    ' variable in the caller's scope. Otherwise, don't.
    someName$ = UCase$(someName$)
    Print someName$
End Sub
firstName$ = "David"
Call PrintName2(firstName$)     ' firstName$ is passed by reference
                                ' by default.
' Output: DAVID
Print firstName$
' Output: DAVID
firstName$ = "David"
Call PrintName2((firstName$))
' Output: DAVID
Print firstName$
' Output: David
```

You can call a sub that expects a single argument by simply entering the sub's name and the argument. If you enclose the argument in parentheses, it gets passed by value to the sub. For example:

```
firstName$ = "David"
PrintName2(firstName$)          ' firstName$ is passed by value.
' Output: DAVID
Print firstName$
' Output: David
PrintName2 firstName$           ' firstName$ is passed by reference.
' Output: DAVID
Print firstName$
' Output: David
```

Executing a sub that takes multiple arguments

When you call a sub that expects multiple arguments, you must enclose those arguments in parentheses when you include the sub in a Call statement, and you must not enclose them in parentheses when you call the sub by simply entering its name followed by its arguments. For example:

```
Dim lastName As String
Sub PrintName3(pronom As String, cognom As String)
   pronom$ = UCase$(pronom$)
   cognom$ = UCase$(cognom$)
   Print pronom$ & " " & cognom$
End Sub
firstName$ = "David"
lastName$ = "LaFontaine"
Call PrintName3(firstName$, lastName$)
Output: ' DAVID LAFONTAINE
firstName$ = "Julie"
lastName$ = "LaFontaine"
PrintName3 firstname$, lastName$
' Output: JULIE LAFONTAINE
```

Specialized subs

LotusScript recognizes four specialized kinds of user-defined sub to allow the automation of set-up and clean-up operations in an application:

- **Sub Initialize**, which executes when the application loads the module in which you defined it
- **Sub Terminate**, which executes when the application unloads the module in which you defined it
- **Sub New**, which executes when you create an instance of the user-defined class in which you defined it
- **Sub Delete**, which executes when you delete an instance of the user-defined class in which you defined it

Sub Initialize

A Sub Initialize lets you perform set-up operations on loading a module. LotusScript automatically executes a Sub Initialize when the application loads the module in which you defined it, performing the operations specified in the sub. You can define only one Sub Initialize per module. The syntax for Sub Initialize is:

```
Sub Initialize
    statements
End Sub
```

By definition, a Sub Initialize is unalterably Private in scope. Its signature can't include a parameter list: LotusScript has no way of passing arguments to a Sub Initialize when it calls it. A Sub Initialize is not subject to the usual restrictions concerning the sorts of statements and directives that a user-defined procedure can contain.

Note Not all implementations of LotusScript support a user-defined Sub Initialize.

Sub Terminate

A Sub Terminate lets you perform clean-up operations when the application unloads a module. As with Sub Initialize, LotusScript automatically executes a Sub Terminate when the application unloads the module in which it is defined, performing the operations specified in the sub. You can define only one Sub Terminate per module. The syntax for Sub Terminate is:

```
Sub Terminate
    statements
End Sub
```

Again, like Sub Initialize, a Sub Terminate is Private in scope, its signature can't include a parameter list, and it is not subject to the usual restrictions concerning the sorts of statements and directives that a user-defined procedure can contain.

Sub New and Sub Delete

Sub New and Sub Delete are special features of user-defined classes. For more information on these subs, see Chapter 5, "User-Defined Data Types and Classes."

Properties

To facilitate the use of object-oriented programming techniques (such as those supported by languages like C++), LotusScript supports a special kind of procedure, the **property**. A property is a language element whose main purpose is to allow the indirect manipulation of variables that you don't want to expose to the application as a whole. To the application, a property looks like a variable to which you can assign and from which you can retrieve a value, but it is actually more than that.

You create a property by defining two procedures, one of which (Property Set) assigns the value of the property to a variable you want to manipulate, and the other of which (Property Get) assigns the current value of that variable to the property. You execute the Property Get procedure by assigning the property a value, and you execute the Property Set procedure by including the property in a statement that uses its value. The application operates on the property (which operates on the variable) rather than on the variable itself. Because Property Set and Property Get are procedures, you can make them perform operations in addition to assigning and retrieving values.

Declaring and defining properties

Essentially the same conventions and restrictions that apply in declaring and defining functions and subs apply in declaring and defining a property: you can define a property at module level or as a member of a user-defined class. Declaring a property before you define it allows you to refer to that property before you actually define it. The circumstances under which you should or should not use the Declare statement to declare a property before referring to it are the same as those that apply in the case of user-defined functions and subs.

The syntax for declaring a property is as follows:

Declare [**Public | Private**] [**Static**] **Property Set** *propertyName* [**As** *dataType*]

and

Declare [**Public | Private**] [**Static**] **Property Get** *propertyName* [**As** *dataType*]

The syntax for defining a property is as follows:

[**Public | Private**] [**Static**] **Property Set** *propertyName* [**As** *dataType*]

    statements

End Property

and

[**Public | Private**] [**Static**] **Property Get** *propertyName* [**As** *dataType*]

    statements

End Property

Element	*Description*
Public, Private	When you declare a property at module level, Public indicates that the application can refer to the property outside the module in which it is defined, as long as that module is loaded; and Private indicates that the property is available only within the module in which it is defined. When you declare a property inside the definition of a user-defined class, Public means that the property is available outside the class definition; and Private means that the property is only available within the class definition. By default, properties defined at module level are Private, and properties defined within a class are Public.
Static	Declares variables defined within the property to be static by default. Static variables retain their values (rather than going out of existence) between calls to the property while the module in which the property is defined remains loaded.
propertyName	The name of the property, which can end in one or another of the LotusScript data type suffix characters (%, &, !, #, @, and $), which determine the data type of the property's return value. You can append a data type suffix character to a property name when you declare it only if you do not include the As *dataType* clause in the declaration.
As *dataType*	Specifies the data type of the property's return value. A property can return a scalar value, a Variant, or an object reference. If you include this clause, *propertyName* cannot end in a data type suffix character. If you omit this clause and *propertyName* doesn't end in a data type suffix character (and isn't covered by an existing Def*type* statement), the property's return value is Variant.

When you define a property, the signatures of the Property Set and Property Get statements must agree as to scope, lifetime of variables, name, and data type.

Using properties

In the following example, the sub KeepGoing uses the property theCube# to manipulate three variables (anInt%, aDouble#, and bigNum#) that are not otherwise referred to directly by the application.

```
%Include "LSCONST.LSS"

Dim anInt As Integer
Dim aDouble As Double
Dim bigNum As Double

Property Set theCube As Double
   anInt% = theCube#
End Property
```

```
Property Get theCube As Double
   aDouble# = anInt% ^ 3
   If aDouble# > bigNum# Then
       bigNum# = aDouble#
  End If
  theCube# = anInt%
End Property

Sub KeepGoing
  Dim goAgain As Integer
  Dim msg As String
  Dim msgSet As Integer
  Dim more As Integer
  goAgain% = TRUE
  msg$ = "Want to go again?"
  msgSet% = MB_YESNO + MB_ICONQUESTION
 ' Prompt the user to enter a number; assign that number to
 ' the property theCube# (by executing Property Set theCube#);
 ' calculate the cube of that number (by executing
 ' Property Get theCube#), assign it to the variable aDouble#,
 ' and compare it to the current value of bigNum#, resetting
 ' the latter if aDouble# is greater. Prompt the user to repeat
 ' the process or quit.
  While goAgain% = True
    ' Execute Property Set theCube# by assigning it
    ' a value. This assigns a value to anInt%.
    theCube# = CInt(InputBox$("Enter an integer:"))
    ' Execute Property Get theCube# by including theCube#
    ' in a Print statement. This assigns a value to aDouble#,
    ' may assign a value to bigNum#, and returns the current
    ' value of anInt%.
    Print theCube# & " cubed = " & aDouble# & "."
    Print bigNum# & " is the biggest cube so far."
    ' See if the user would like to do all this again or quit.
    more% = MessageBox(msg$, msgSet%)
      If more% = IDNO Then
         goAgain% = FALSE
      End If
  Wend
  Print "All Done."
End Sub
```

```
Call KeepGoing

' Output: The user types 3 and selects Yes, then
          4 and selects Yes, then 2 and selects No.
' 3 cubed = 27.
' 27 is the biggest cube so far.
' 4 cubed = 64.
' 64 is the biggest cube so far.
' 2 cubed = 8.
' 64 is the biggest cube so far.
' All Done.
```

As the following example shows, you could just as well perform the same operations as in the preceding example using a sub and a function instead of a property:

```
%Include "LSCONST.LSS"

Dim anInt As Integer
Dim aDouble As Double
Dim bigNum As Double

Sub SetTheCube
   anInt% = CInt(InputBox$("Enter an integer:"))
End Sub

Function GetTheCube(anInt As Integer) As Double
   aDouble# = anInt% ^ 3
   If aDouble# > bigNum# Then
       bigNum# = aDouble#
  End If
  GetTheCube# = anInt%
End Function

Sub KeepGoing
  Dim goAgain As Integer
  Dim msg As String
  Dim msgSet As Integer
  Dim more As Integer
  goAgain% = TRUE
  msg$ = "Want to go again?"
  msgSet% = MB_YESNO + MB_ICONQUESTION
```

```
  While goAgain% = True
    Call SetTheCube
    Print GetTheCube#(anInt%) & " cubed = " & aDouble# & "."
    Print bigNum# & " is the biggest cube so far."
    ' See if the user would like to do all this again or quit.
    more% = MessageBox(msg$, msgSet%)
      If more% = IDNO Then
         goAgain% = FALSE
      End If
  Wend
  Print "All Done."
End Sub

Call KeepGoing

' Output: The user types 3 and selects Yes, then
'         4 and selects Yes, then 2 and selects No.
' 3 cubed = 27.
' 27 is the biggest cube so far.
' 4 cubed = 64.
' 64 is the biggest cube so far.
' 2 cubed = 8.
' 64 is the biggest cube so far.
' All Done.
```

What properties are really good for is manipulating protected variables, that is, Private members of a user-defined class to which the application has no direct access. See the following chapter, “User-Defined Data Types and Classes,” for a description of user-defined classes and how to use properties to manipulate their member variables.

Chapter 5
User-Defined Data Types and Classes

This chapter describes two kinds of custom data structure that you can define in LotusScript. Each can hold data of different types in a single data structure. This chapter covers the following topics:

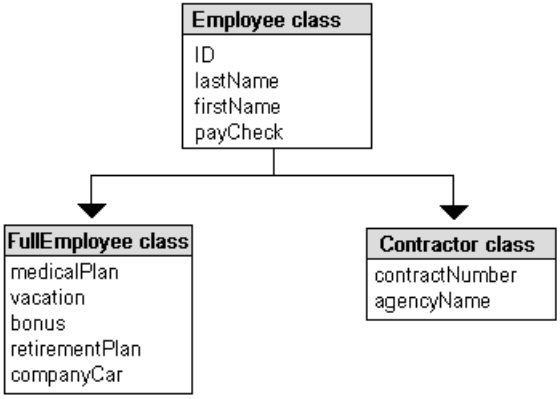
- Comparison of user-defined data types and classes
- User-defined data types
- Introduction to classes
- Creating base classes
- Creating, managing, and deleting objects
- Creating derived classes
- Creating arrays and lists of classes

Comparison of User-Defined Data Types and Classes

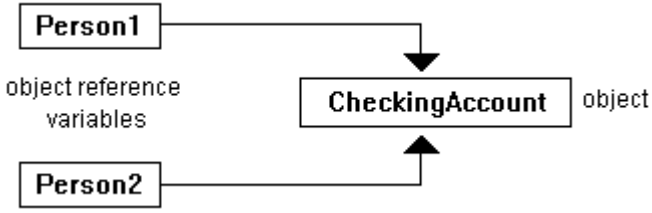
User-defined data types and classes are data structures that both hold data of different types, but there are some subtle differences between the two. User-defined data types are a common feature in BASIC programming and are used to support database, file read / write, and print operations. Classes are common to object-oriented programming and are used to represent objects whose data can be protected, initialized, and accessed by a specific set of procedures.

User-defined data types and classes can both contain multiple variables of different data types. Unlike user-defined data types, classes can also contain procedures (properties and methods) that operate on those variables.

You can extend a class but not a user-defined data type. That is, you can derive new classes (called **derived classes**) from an existing class (called a **base class**), where the derived classes inherit from the existing (base) class. For example, you could extend an Employee class by creating a FullEmployee class to represent full-time employees and a Contractor class to represent temporary employees. Both the FullEmployee class and the Contractor class share common data (ID, lastName, firstName, payCheck) provided by the Employee class.



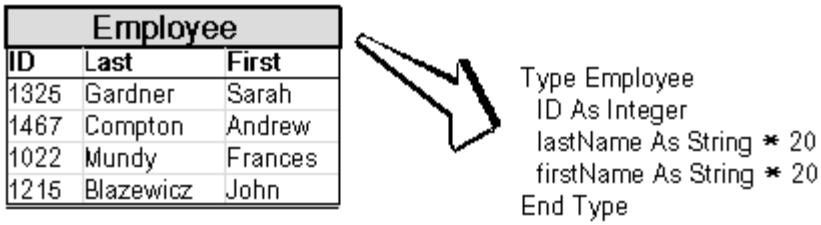
Another important difference between user-defined data types and classes is that a variable of a user-defined data type holds actual data, while a class's object reference variable points to an object's data stored in memory. For example, Person1 and Person2 can be object reference variables that point to the same CheckingAccount object. This flexibility allows two different people to access the same checking account.



In general, you create a user-defined data type for operations that don't need properties and methods. For example, you might create a data type named Coordinates that contains member X and Y coordinates in order to perform simple file read / write operations. In most other cases, you will want to create classes.

User-Defined Data Types

You can save time and effort when you build applications by creating user-defined data types. A **user-defined data type** lets you group data of different types in a single variable. This data type can contain any kind of related information you want to store and use together, such as personnel information, company financial information, inventory, and customer and sales records. The following illustration shows how you could create an Employee data type that contains three member variables (ID, lastName, and firstName) to hold database records of employee information:



Defining user-defined data types

You define a user-defined data type using this syntax:

[**Public** | **Private**] **Type** *typeName*

member variable declarations

End Type

Element	*Description*
Public, Private	Public specifies that the data type is accessible outside the module in which it is defined. Private (default) specifies that the data type is accessible only within the module in which it is defined.
typeName	The name of the data type.
member variable declarations	Declarations for members of the type. Member variables can hold scalar values, Variants, fixed arrays, or other user-defined data types. A member variable declared as Variant can hold fixed or dynamic arrays, a list, or an object reference, in addition to any scalar value. Declarations cannot include Const statements.

While member variable declarations resemble those of local variables declared in a function, LotusScript allocates space for them only when an application creates the user-defined data type. When this happens, LotusScript allocates space for all the member variables at the same time.

Declaring a variable of a user-defined data type

After you define a user-defined data type, you can declare a variable of this type:

```
Dim President As Employee ' Create a single employee record.
```

If you want to hold data from many database records, you can declare an array of variables of this type:

```
Dim Staff(10) As Employee ' Create an array of ten employee records.
```

Referring to member variables

You use dot notation (*object.memberVariable*) to refer to member variables, and you use an assignment statement to assign values to member variables:

```
President.ID = 42
President.lastName = "Wilkinson"
President.firstName = "May"
```

You can refer to the elements of a member variable that is an array or list:

```
Staff(1).ID = 1134
Staff(1).lastName = "Robinson"
Staff(1).firstName = "Bill"

Staff(2).ID = 2297
Staff(2).lastName = "Perez"
Staff(2).firstName = "Anna"
```

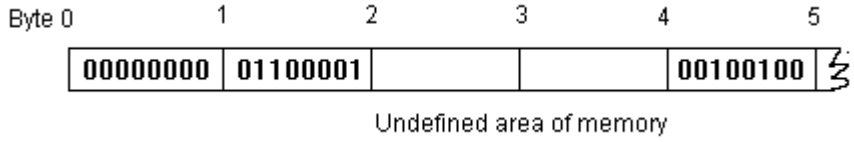
You can retrieve data from member variables by assigning a member variable value to a variable or printing the value of a member variable:

```
Dim X As String
X$ = Staff(2).lastName
Print X$              ' Prints Perez.
```

Conserving memory when declaring member variables

Members of a user-defined data type are not necessarily stored in consecutive bytes of memory. Sometimes there is wasted space in the data type because, due to computer hardware requirements, LotusScript stores variables of certain data types on architecture-dependent boundaries. For example, LotusScript stores an Integer on a 2-byte boundary and a Double on an 8-byte boundary.

Because of these boundary requirements, you can waste space in a user-defined data type when you define its members. For example, if the first member variable is an Integer value (2 bytes) and the second member variable is a Long value (4 bytes), the Integer is aligned on a 2-byte boundary and the Long is aligned on a 4-byte boundary, which results in an undefined area of memory.



You can use data space efficiently by declaring members with the highest boundary first (Variant is biggest at 16 bytes), and those with the lowest boundary last (fixed-length Strings). This is especially important because wasted space in the definition of a user-defined data type becomes wasted space in every variable of that user-defined data type.

Here is an example of a well-aligned type:

```
Type T1
  m1 As Variant     ' 16 bytes
  m2 As Double      '  8 bytes
  m3 As Long        '  4 bytes
  m4 As String      '  4 bytes
  m5 As Integer     '  2 bytes
  m6(10) As Integer '  2 bytes
  m7 As String * 30 '  1 byte
End Type
```

LotusScript stores a variable of a user-defined data type on a boundary equal to the size of its largest member. For example, the following script, continued from above, shows how each variable of user-defined data type T1 is aligned on a 16-byte boundary:

```
Type T2
  m1 As T1 ' 16-byte boundary; T1's largest member boundary is 16.
  m2(3) As Long     ' 4 bytes.
End Type
```

Keep the following issues in mind when you declare member variables:

- A fixed-length string is not aligned on any boundary.
- A fixed array is aligned on the boundary of its declared data type.
- The order for data types that align on the same boundary is not important. For example:

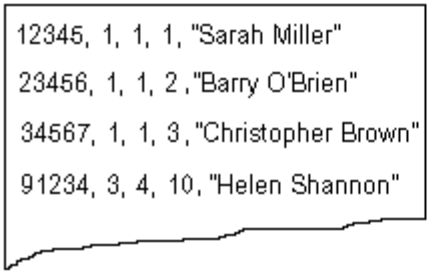
  ```
  Dim x As Long
  Dim y As String
  ```

  is as efficient as

  ```
  Dim y As String
  Dim x As Long
  ```

Working with data stored in files

You often create user-defined data types to work with data stored in files. For example, the script below and following illustration read a sample ASCII file that contains employee parking information into an array of user-defined data types:



A comma-delimited text file

```
Type RecType
  empID As Double                ' Employee ID
  theSection As Integer          ' Car parking section
  theSpace As Integer            ' Designated parking space
  theFloor As Integer            ' Car parking level
  employee As String             ' Employee name

End Type

' Dynamic array sizes to fit the lines in the file.
Dim arrayOfRecs() As RecType

Dim txt As String
Dim fileNum As Integer
Dim counter As Integer
Dim countRec As Integer
Dim found As Integer

fileNum% = FreeFile             ' Get a file number to open a file.
counter% = 0
Open "c:\myfile.txt" For Input As fileNum%
Do While Not EOF(fileNum%)
    Line Input #fileNum%, txt$  ' Read each line of the file.
    counter% = counter% + 1     ' Increment the line count.
Loop
Seek fileNum%, 1                ' Pointer to beginning of file
' Since file has counter% number of lines, define arrayOfRecs to
' have that number of elements.
ReDim arrayOfRecs(1 To counter%)
```

```
' Read the file contents into arrayOfRecs.
For countRec% = 1 to counter%
   Input #fileNum%, arrayOfRecs(countrec%).empID, _
     arrayOfRecs(countrec%).theSection, _
     arrayOfRecs(countrec%).theSpace, _
     arrayOfRecs(countrec%).theFloor, _
     arrayOfRecs(countRec%).employee
Next
Close fileNum%
' Elicit an employee's name and look for it in arrayOfRecs.
ans$ = InputBox$("What's your name?")
found% = False
For x% = 1 To counter%
   If arrayOfRecs(x%).employee = ans$ Then
       found% = True
       Print "Greetings, " & ans$ & "."
       Exit For
   End If
Next
If found% = False Then Print "No such employee.
```

Classes

You can build object-oriented applications by creating classes. A **class** is a data type that restricts access to its data to a set of procedures. These procedures control the ways that an instance of a class (an **object**) is initialized, accessed, and finally deleted when it is no longer needed.

A class lets your application model real objects, their attributes, and their behaviors. For example, an air traffic-control system creates a flight class, a car rental system creates a car class, and a bank's automated teller system creates an account class. For each class, you define its members: variables, properties, and subs and functions (also called **methods**). Typically, you can retrieve and assign values to an object's properties. Methods perform operations on the object.

Class	*Properties*	*Methods*
Flight	GateNumber FlightNumber InAir OnGround	TakeOff Land DelayFlight CancelFlight
Car	LicensePlate DriverLicense RentalDate	ServiceCar TransferLocation
Account	CustomerNumber Balance AccountNumber	WithdrawCash DepositMoney MoveMoney

In a script, you can declare a variable to refer to an instance of the object's class. The variable is an **object reference variable**. Each class defines the data used by instances of the class and defines a set of properties and methods that apply to the class.

Benefits of classes

Classes offer several features that can simplify your application programming:

- Classes provide more functionality than any other LotusScript data type. A class can hold any type of data, including instances of the class being defined.
- Classes are self-contained so it's easy to use the same class in another application. For example, a File class that provides general file input/output functions can be shared with other applications. Reusing classes reduces the time to design, write, and test your application, increases the likelihood that your scripts are correct, and saves time when you need to update scripts.
- You can simplify the programming interface to your application by creating classes that call the Windows® API (Application Programming Interface), or any C-API. Users of the class work only with the class's member variables, properties, and methods, and do not require knowledge of Windows or C-API programming.
- You can build class libraries (a collection of classes) to allow other application developers to use your classes without allowing them to modify the class scripts. To do this, you compile classes into .LSO files and provide access via the Use statement.
- You can use classes to build tools for your applications. For example, you can create a class that allows your application to access the spelling checker and dictionary that come with most Lotus products.

Types of classes

You can create two types of LotusScript classes:

- A **base class** defines common member variables, properties, and methods that can be inherited by other classes.
- A **derived class** extends and elaborates an existing base class. A derived class has direct access to all members of the existing base class. However, the derived class can add new member variables, properties, and methods, and it can redefine properties and methods from the base class, while leaving the base class unchanged. For example, you could create SavingsAccount and CheckingAccount classes based on an Account class.

Base classes

You define a base class using this syntax:

[**Public** | **Private**] **Class** *className*

classBody

End Class

Element	*Description*
Public, Private	Public specifies that the class is accessible outside the module in which it is defined. Private (default) specifies that the class is accessible only within the module where the class is defined.
className	The name of the class.
classBody	Declares member variables, and declares and defines properties and methods. Member variables can have any data type LotusScript supports, and can be object reference variables of the class being defined. Methods can be functions and subs, including Sub New, which initializes class objects, and Sub Delete, which deletes class objects. You cannot declare a class member as Static.

Declaring member variables

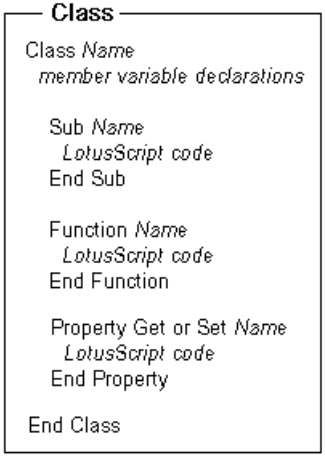
While class member variable declarations resemble those of local variables declared in a function, LotusScript allocates space for them only when an application creates an instance of a class. When this happens, LotusScript allocates space for all the class's member variables at the same time.

You can define a class using any mixture of data types for member variables, including object references to the class being defined:

```
Class MyClass
  myText As TextBox          ' Sample product object reference
  i As Integer               ' Integer
  myList List As String      ' List of strings
  myRef As MyClass           ' Reference to an object of this class
End Class
```

Defining member properties and methods

Properties and methods are tied to their class and can be used only with an object belonging to that class. You define properties and methods inside the Class statement.



- A **property** is a pair of functions that you can manipulate the way you would a variable (see Chapter 4, “Procedures: Functions, Subs, and Properties”). You use properties to manipulate protected variables, that is, Private members of a user-defined class to which the application has no direct access.
- A **method** is a sub or function that performs operations on objects.

The following Stack class uses several properties and methods to perform simple push and pop operations on a stack data structure.

```
Class Stack
  Private idx As Integer
  Stack List As Variant
  Public stackName As String
  Private Sub CheckStack  ' Sub is visible only within the class.
    If idx% = 0 Then Error 999
  End Sub

  Sub New
    idx% = 0        ' Initialize idx.
  End Sub

   Private Property Set topValue As Variant
     CheckStack
     Stack(idx%) = topValue  ' Set the top value on the stack.
   End Property

   Private Property Get topValue As Variant
     CheckStack
     topValue = Stack(idx%)  ' Get the top value on the stack.
   End Property

   ' Same as Get for topValue.
   Function Top
      Top = topValue   ' Call the topValue Get method.
   End Function

   Sub Push(v)         ' Push a value on the stack.
     idx% = idx%+1
     topValue = v
   End Sub

   Function Pop        ' Pop a value off the stack.
     Pop = topValue
     Erase Stack(idx%)
     idx% = idx%-1
   End Function

   ' Read-only property. There is no Set for Count.
   Property Get Count
     Count = idx%       ' Count the values on the stack.
   End Property

End Class
```

```
Dim St As New Stack
Call St.Push("An item on the stack")
Call St.Push("Another item on the stack")
Print "# of items on the stack is ";St.Count
Print "TopValue is ";St.Top
```

Declaring Sub New and Sub Delete (initializing and deleting objects)

Within a class definition you can define two special subs, Sub New and Sub Delete, that let your application initialize and delete objects. **Sub New** is a sub that LotusScript executes automatically when an object is created. **Sub Delete** is a sub that LotusScript executes automatically when an object is deleted. You use Sub New to initialize a newly created object. You use Sub Delete to perform termination housekeeping on the object. LotusScript executes these subs automatically; you cannot call them explicitly.

For example, you could create a File class that uses Sub New to open a file and Sub Delete to close a file. Similarly, you could create a PrintJob class that uses Sub New to start a new page, align text, and set the left and right margins, and that uses Sub Delete to terminate the print job.

Keep the following issues in mind regarding Sub New and Sub Delete:

- A class can have one Sub New and one Sub Delete.
- Sub New executes automatically when LotusScript executes a Dim statement with the New keyword, or executes a Set statement, referring to the class for which the Sub New is defined. Sub Delete executes automatically when the object for which it is defined is deleted.
- Sub New and Sub Delete are always Public; you cannot declare them as Private.

You declare Sub New and Sub Delete as part of the definition of a class. You create a Sub New by defining a sub named New and the parameters to initialize the newly created object. You create a Sub Delete by defining a sub named Delete, without specifying parameters.

Sub New in the following script initializes the member variables of the CustomerAccount object. The Set statement that creates a new Account object also passes three arguments required by the Sub New for the Account class. Sub New assigns the values of the arguments to the three member variables of the newly created object: balance@, acctNum&, and customerNum&.

```
Class Account
  balance As Currency
  acctNum As Long
  customerNum As Long

' Declare Sub New.
  Sub New (newBal As Currency, newAcctNum As Long, newCustNum As Long)
     balance@ = newBal@
     acctNum& = newAcctNum&
     customerNum& = newCustNum&
     Print "New Parms=";balance@, acctNum&, customerNum&
  End Sub

' Declare Sub Delete.
  Sub Delete
     Print "Deleting account record for customer: ";customerNum
  End Sub

End Class
'.....
Dim CustomerAccount As Account

' Create the object.
Set customerAccount = New Account(1234.56, 10001991, 5412)

Delete customerAccount           ' Explicitly delete the object.
```

About Public and Private class members

When you define a class, you can make members Public (so members can be referred to by statements outside the class definition) or Private (so members can be referred to only by properties and methods defined in that class). If you don't specify otherwise, member variables are Private by default; and properties, subs, and functions are Public by default.

It is good programming practice to keep class member variables Private, and to use Public properties and methods to manipulate the private data stored in member variables. Keeping member variables Private is often called **data hiding** or **encapsulation** because private data is hidden from subs and functions defined outside the class. Keeping properties and methods Public provides public access to the class's users.

Referring to class members inside a class's scope

A class's scope is everything within the Class...End Class statement. Within a class's scope, class members are immediately accessible to all of that class's properties and methods.

You can refer to an individual member of a class by using its name. For example, to print the value in a member variable called employeeName$, you use the following statement:

```
Print employeeName$
```

Within a property or method, you can use the keyword Me to access the class's definition. This is particularly useful in Sub New when you are assigning external values to member variables. For example, you can use **`Me.memberVariable = externalValue`** to assign a value. You can also use Me when you need to do the following:

- Refer to a class member that has the same name as a local variable. For example, if a property or method contains a local variable X, and X is also the name of a class member, use Me.X within the method to refer to the member X.
- Pass a reference to the class as an argument to a procedure.

You must use Me to access class members that have the same names as LotusScript keywords. For example, the following class definition uses Me to refer to a class member that is a keyword.

```
Class MyObject
   ' ...
   ' Reserved keyword Read is used here to name a function.
   Function Read
      Dim x As Integer           ' Status of operation.
       ' ....
       ' Me is required to refer to the function named Read.
       Me.Read = x%
   End Function
   ' ...
End Class
```

Creating, Managing, and Deleting Objects

You use object reference variables to create, manage, and delete objects. An **object reference variable** is different from other variables because it is associated with an instance of a class (that is, an object). It has the data type of a class and, like other variables, is a named area in storage. However, unlike other variables, the value stored in the area named by an object reference variable is not the object itself. Instead, the object (and the data it consists of) is stored elsewhere. The value stored in an object reference variable is a 4-byte pointer to the object data, called an **object reference**. LotusScript uses this pointer to access the object data. When you assign a value to an object reference variable, you associate (or bind) the object reference to the object.

Working with object reference variables

When you create an instance of a class, you must explicitly declare an object reference variable. That is, you create the object, create the object reference variable, and assign an object reference to the variable. The object reference points to the object. When an object is created, its member variables are initialized, each to the initial value for the data type of the member. For example, a member of data type Integer is initialized to 0. If a member is itself a user-defined data type or a class, it is initialized by initializing its member variables.

You can create an object reference without creating an object with the following syntax:

```
Dim x As ClassName
```

Because the variable you declare contains a reference to an object that does not yet exist, the variable is initialized to the value NOTHING.

Creating objects

After defining a class, you create objects using the LotusScript New keyword. You can use the New keyword with either the LotusScript Dim or Set statement. You can use the keyword New in the declaration that declares an object reference variable. If you use the New keyword when you declare the object reference variable, the declaration creates an object and assigns to the variable a reference to the newly created object.

- To create a new object and assign a reference to that object in a variable that you are declaring, use the Dim statement with the following syntax:

  Dim *objRef* **As New** *className*[(*argList*)]

- To create a new object and assign a reference to it if you have already declared an object reference variable (with a Dim statement without the New keyword), use the Set statement with the following syntax:

  Set *objRef* = **New** *className*[(*argList*)]

You can't use the New keyword to declare an array of object reference variables or a list of object reference variables.

In the following example, X can hold only references to Demo objects, or else the value NOTHING. It is initialized to NOTHING.

```
Class Demo
  ' ...
End Class

' Declare an object reference variable X of the class
' Demo, create an instance of that class, and assign X
' a reference to the new Demo object.
Dim X As New Demo

Dim DemoArray(10) As Demo ' Array of object reference variables
Dim DemoList List As Demo ' List of object reference variables
```

LotusScript initializes each element of DemoArray to NOTHING. However, since a list has no elements when it is declared, LotusScript does not initialize the elements in DemoList. Each element of DemoArray, and each element of DemoList, when created, can hold either the value NOTHING or a reference to a Demo object, for example:

```
Set DemoArray(0) = New Demo
```

Using the Set statement

You can create an instance of a class by using a Set statement that includes the New keyword and a variable that was previously declared as an object reference variable for that class.

The Set statement is a kind of assignment statement used only to assign values (object references) to object reference variables. You can't use it to assign values to any other kind of variable. At the same time, you cannot use the common assignment statement, using the equal sign (=), with or without the Let keyword, to assign values to object reference variables.

You can assign a reference to a newly created object to an array element or a list element. Continuing from the previous example:

```
Dim Z(10) As Demo    ' Declare an array of object reference variables.

Dim A List As Demo   ' Declare a list of object reference variables.

Set Z(1) = New Demo  ' Assign Z(1) a reference to the created object.

'Assign a list element a reference to the created object.
Set A("ITEM01") = New Demo
```

You can assign an existing object reference to another variable using the Set statement without the New keyword, as in the following example:

```
Class Customer
 ' ...
End Class
' Declare object reference variable C, create a Customer object,
' and assign C a reference to the new Customer object.
Dim C As New Customer

'Declare object reference variable myArray and initialize
'all elements of MyArray to NOTHING.
Dim myArray(10) As Customer

Dim dTwo As Customer  ' Object reference is set to NOTHING.

Set dTwo = myArray(1) ' Assign the myArray(1) value, NOTHING, to DTwo.

Set myArray(1) = C  ' myArray(1) and C refer to the same Customer.

Set dTwo = myArray(1)  ' Now dTwo also refers to the same Customer.

Set myArray(1) = NOTHING ' Set the object reference to NOTHING.
' Assign myArray(1) a reference to a new Customer object.
Set myArray(1) = New Customer
' Assign dTwo a reference to a new customer object. Now, variables
' C, myArray(1), and dTwo each refer to different Customer objects.
Set dTwo = New Customer
```

Note that an assignment using Set does not copy an object. The assigned value is a reference to an object, not the object itself. The value stored in an object reference variable is a pointer to the data that makes up the object. Set copies the reference into the target variable.

Using Variants to hold object references

You can assign an object reference to a variable of type Variant. In the following script, the variable anyFruitV holds a reference to Fruit objects and is of type Variant. The script executes when the user clicks a Notes button.

```
Class Fruit
     Sub PrintColor
          MessageBox ("I have no color.")
     End Sub
End Class
```

```
Class Banana As Fruit
     Sub PrintColor
          MessageBox ("I'm yellow.")
     End Sub
End Class

Class Grape As Fruit
     Sub PrintColor
          MessageBox ("I'm purple.")
     End Sub
End Class

Sub Click(Source As Button)      ' Sample Notes product object.
     Dim myFruit As New Fruit
     Dim myBanana As New Banana
     Dim myGrape As New Grape

     Dim anyFruitV As Variant

     Set anyFruitV = myFruit
     anyFruitV.PrintColor

     Set anyFruitV = myBanana
     anyFruitV.PrintColor

     Set anyFruitV = myGrape
     anyFruitV.PrintColor
End Sub
```

Initializing member variables

Sub New is automatically called when LotusScript executes a Dim or a Set statement with the New keyword and creates an instance of that class. You can use a class's Sub New to initialize member variables, or you can choose to initialize variables using Property Get and Property Set. You can specify parameters so that arguments can be passed to Sub New.

Referring to class members outside of a class's scope

Outside a class's scope, you can refer only to its Public members. By default, a class's member variables are Private and its properties, subs, and functions are Public. You use dot notation to refer to Public class members.

Referring to Public class members

Outside a class's scope, you can access only its Public members. For example, you can access the member variables balance@ and custName$ in the Customer class.

```
Class Customer
  Public custName As String
  Public balance As Currency

  Sub CheckOverdue
    If balance@ > 0 Then
      Print "Overdue balance"    ' Send an overdue letter.
    End If
  End Sub
End Class

Dim X As New Customer
Dim newBal As Currency

' This is a legal statement, because custName is Public.
X.custName$ = "Acme Corporation"
X.balance@ = 14.92               ' Balance@ is Public.

' Assigns the value of the Public member variable balance
' to the variable newBal@.
newBal@ = X.balance@
Print X.balance@; newBal@        ' Prints 14.92  14.92
```

To check for an overdue balance, you can call the Public sub CheckOverdue as in the following example:

```
Dim Y As Customer
Set Y = X
Y.CheckOverdue     'Prints "Overdue balance"
Print Y.balance@; X.balance@     ' Prints 14.92  14.92
```

Referring to members of an object

You can use the With statement as a quick way to access class members of a given object. You can also use the With statement to test expressions using an object's members. The syntax of With is:

With *objectRef*

[*statements*]

End With

Element	*Description*
objectRef	An expression whose value is a reference to an object. For example, *objectRef* can be a function call that returns an object reference or a Variant that contains an object reference.
statements	One or more statements. The With statement itself may be nested up to 16 levels.

The following example uses the With statement to refer to members of an object using a dot to represent the object name (startEmp).

```
Class Employee
   Public empName As String
   Public newName As String

   '  Sub GetName prompts for and accepts input to newName.
   Sub GetName
      newName$ = InputBox$("Enter name:" , "New Name" )
   End Sub
End Class

Dim startEmp As New Employee
'  Sub SetEmp puts information into the new employee object.
Sub SetEmp (E As Employee)
   With E
      Call .GetName    ' Prompts for input to startEmp.newName$.
      .empName$ = .newName$
   End With
End Sub
Call SetEmp(startEmp)
```

Outside the With statement, you need to specify the entire reference. For example:

```
Employee.empName$ = .newName$
```

Testing object references

You use the Is operator to compare object references and to test object reference variables for the value NOTHING. When you use the Is operator to compare two object references, the expression evaluates to True (-1) if they refer to the same object, or if both have the value NOTHING. Otherwise it evaluates to False (0).

The following example illustrates how to test object references:

```
Class MyClass
  ' ...
End Class

Dim x As MyClass
Dim y As MyClass
Dim z As New MyClass
Dim other As New MyClass

Set x = z
If (x Is z) Then Print "Both x and z refer to the same object."
If (y Is NOTHING) Then Print "y is NOTHING. It refers to no object."
If (z Is other) Then _
 Print "This should not print; z and other are different objects."
End If
```

You can also use the Is operator in a flow of control statement, for example in a Do statement:

```
Dim a As New MyClass, b As MyClass
' ...
Do While b Is NOTHING            ' The condition b is NOTHING.
                                 ' Condition is either True or False.
' ...
  Set b = a
Loop
```

Deleting objects

You define a Sub Delete to specify the procedure that LotusScript is to execute just before it deletes an object of the specified class. You can use the Delete statement to explicitly delete objects, or you can let LotusScript delete the object automatically when it is no longer needed.

Sub Delete

A class's Sub Delete is called when LotusScript deletes an object of that class. Sub Delete itself does not actually delete the object — it performs termination housekeeping before the system reclaims the object's memory space so that it may be used to hold new objects. Sub Delete receives no parameters and returns no value. For more information, see "Declaring Sub New and Sub Delete (initializing and deleting objects)," earlier in this chapter.

Deleting an object using the Delete statement

When you use the Delete statement, LotusScript deletes the object even if one or more variables contain references to the object. All object reference variables that contain references to the deleted object are automatically assigned the value NOTHING, and you can no longer refer to the object's members.

In the following example, the variables anObj and otherObj are set to NOTHING. You can reuse these variables because they are still valid references; they simply contain NOTHING.

```
Class DemoObject
  Sub New
    Print "New"
  End Sub

  Sub Delete
    Print "Delete"
  End Sub
End Class

Dim anObj As New DemoObject
Dim otherObj As DemoObject
Set otherObj = anObj     ' Make Other refer to the same object.
Delete anObj             ' Set all the object's references to NOTHING.

If ( (anObj is NOTHING) And (otherObj is NOTHING)) Then _
  Print "Both anObj and otherObj are now NOTHING"
```

Managing memory for objects

LotusScript automatically manages the memory associated with objects you create by tracking all references to the objects. LotusScript also automatically frees the memory for objects by deleting them when no variables refer to the objects.

Here is how LotusScript tracks references to objects: When you create an object, LotusScript assigns a reference to the object and sets the object's reference count to 1. Whenever you assign an object reference for that object to a variable, LotusScript increments the reference count by 1. When an object reference is no longer needed, such as when an object reference variable goes out of scope, LotusScript decrements the object's reference count by 1. When the reference count reaches 0, no variables contain references to the object so LotusScript automatically deletes the object and frees its memory.

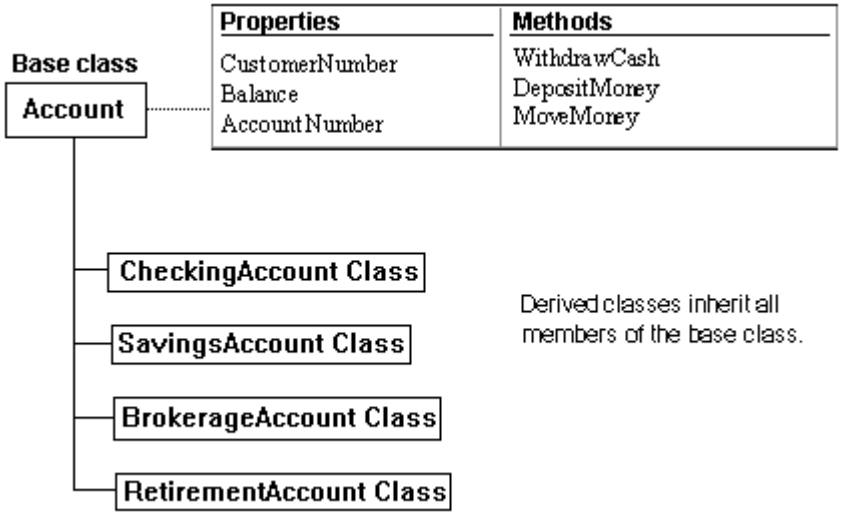
In the following example, LotusScript deletes objects when the reference count returns to 0.

```
Class DemoObject

  Sub New
    Print "New"
  End Sub

  Sub Delete
    Print "Delete"
  End Sub

End Class

  Sub MyDemo
  ' localObject reference count is set to 1.
    Dim localObject As New demoObject
    If (Not (localObject Is NOTHING)) Then _
      Print "In MyDemo sub and localObject exists."
  End Sub

Print "About to call MyDemo."
Call MyDemo
' Sub MyDemo is now out of scope...
' so the reference count is 0 and the object is deleted.
Print "Returned from MyDemo. Delete already ran."
```

Derived Classes

Instead of always creating classes from scratch, you can derive a new class from a previously defined class. Usually you do this when an existing class provides members that the new class can use, or when you want to extend or embellish existing class properties and methods. Deriving new classes from existing classes is called **inheritance**. The new class is called the derived class because it inherits, and has direct access to, all Public and Private members of the existing base class.

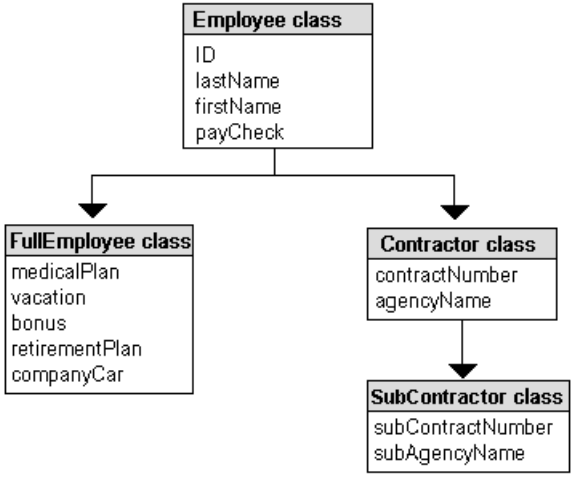
For example, suppose you want to create derived classes called CheckingAccount, SavingsAccount, BrokerageAccount, and RetirementAccount based on an existing Account class. Because the derived classes can access all existing properties and methods for the Account class, such as AccountNumber, Balance, and DepositMoney, you can reuse all Account class scripts in the new classes.



You can define new member variables, properties, and methods in a derived class to add operations that the derived classes can use. For example, you can add BuyStock and SellStock methods to the BrokerageAccount class.

A property or method defined in a base class is accessible in the derived class. You can also modify the behavior of the base class properties and methods used by the derived class. This is called **property overriding** and **method overriding**. There can be many reasons to override properties and methods. For example, the derived class may need to extend a base class's property to perform specialized data validation, or the derived class may need to replace a method entirely (for example, to substitute a different calculation for a currency exchange). You can override the Account class WithdrawCash method (shown in the preceding illustration) so that the RetirementAccount class can use it, for example, to handle special regulations for withdrawals from retirement accounts.

A derived class can serve as the base class for another derived class. For example, the following illustration shows how the Contractor class, which is derived from the Employee class, serves as the base class for the Subcontractor class. The Subcontractor class has access to the members of both the Contractor class and the Employee class.



A derived class has the same scope as its base class, except that a derived class cannot access the Sub Delete of its base class.

Defining derived classes

Use this syntax to define a new class based on an existing class:

[**Public** | **Private**] **Class** *className* **As** *baseClass*

classBody

End Class

Element	*Description*
Public, Private	Public makes the derived class accessible outside the module in which it is defined. Private (default) makes the derived class accessible only within the module in which it is defined.
className	The name of the derived class.
baseClass	The name of the base class from which this class is derived.
classBody	Member variables can have any data type LotusScript supports and can be object reference variables of the class being defined. You can also specify properties, functions, and subs, including Sub New, which initializes class objects, and Sub Delete, which deletes class objects. You cannot declare a class member as Static.

Here is a derived class called MyClass2 that uses the base class MyClass1:

```
Class MyClass1                    ' Base class.
      a As Integer
      Public c As Integer
      '...
End Class

Class MyClass2 As MyClass1      ' Class derived from MyClass1.
      b As Integer
      Public d As Integer
      '...
End Class
Dim x As New MyClass2     ' Object x has members a%, b%, c%, and d%.
x.c% = 12
x.d% = 35
'...
```

Defining derived class members

You override a base class property by redefining a property in the derived class. You override a method by redefining a sub or function in the derived class. The signature of the overriding method must be identical to that of the base class method. That is, the parameters to the method in the derived class must match exactly the parameters to the method in the class in which it was originally defined.

The following example creates two classes that are related by inheritance. The script declares a base class named Fruit, and then declares Apple and Banana to be new classes derived from the Fruit class. The Apple and Banana classes inherit all of the Fruit class's variables (weight and color) and the Prepare sub.

The Prepare sub is intentionally left blank in the base class. It provides general access and allows itself to be overridden and extended in the derived classes so that you can access Apple or Banana functionality via a Fruit sub. Both derived classes override the base class's Prepare sub. The Apple class substitutes a Core sub and the Banana class substitutes a Peel sub.

```
Class Fruit

weight As Single
color As String
  Sub New(w As Single, c As String)
    weight! = w!
    color$ = c$
  End Sub
```

```
  Sub Prepare
    ' Assume that each derived class will override
    ' the Prepare method.
    ' Print a message...
      Print "The Fruit class's Prepare sub doesn't do anything."
  End Sub

End Class

Class Apple As Fruit ' Derive the Apple class from the Fruit class.
  seedCount As Integer
  variety As String
   Sub Core          ' Add a Core sub to the Apple class.
     If (weight! > 5) Then  ' You can access base class members.
     Print "This apple core method is for apples of 5 lbs. or less."
     Exit Sub
     End If
     '...
     Print "The ";weight!;" lb. ";color$;" "; variety$; _
        " apple is cored."
   End Sub

   Sub New(w As Single, c As String, v As String, s As Integer), _
      Fruit (w!,c$)
    Variety$ = v$      ' Initialize the variety.
    SeedCount% = s%    ' Initialize the number of seeds.
   End Sub

   Sub Prepare
      Core     ' To prepare an apple, you core it.
   End Sub
End Class

Class Banana As Fruit  ' Banana class is derived from the Fruit class.
   Sub Peel  ' Add a peel method to the Banana class.
     '.
     Print "The ";weight!;" lb. ";color$;" Banana is now peeled."
   End Sub
  Sub New(w As Single, c As String)
    '...
  End Sub

  Sub Prepare
    Peel               ' To prepare a banana, you peel it.
  End Sub
End Class
```

Extending Sub New for derived classes

You can define Sub New for a derived class to augment the definition of its base class's Sub New. Sub New for a derived class must provide the base class Sub New with its expected parameters.

The parameter list for the base class's Sub New can be a subset of the parameter list for the Sub New of the derived class. You can pass any expression, including a constant or a variable declared at module level, as an argument to the base class's Sub New. You can omit the arguments for the base class's Sub New if the arguments for the derived class Sub New and the base class Sub New are the same.

You extend Sub New for a derived class using the following syntax:

Sub New [(*paramList*)] [, *baseClass* (*baseArgList*)]

[*statements*]

End Sub

Element	*Description*
paramList	A comma-separated list of parameter declarations for Sub New. Use this syntax for each parameter declaration: [**ByVal**] *paramName* [( ) | **List**] [**As** *dataType*] ByVal passes *paramName* by value: that is, the value assigned to *paramName* is a local copy of a value in memory, rather than a pointer to that value. *paramName*() is an array variable; List identifies *paramName* as a list variable; otherwise, *paramName* can be a variable of any of the other data types that LotusScript supports. As *dataType* specifies the variable's data type.
baseClass	An identifier of the class from which the class is derived. *baseClass* must be the same as the *baseClass* in the Class statement for the derived class.
baseArgList	A comma-separated list of arguments for the Sub New of the base class. These arguments are passed to the Sub New of the *baseClass*. Specify this argument list if the arguments to Sub New of the base class do not match those for Sub New of the derived class in number and / or data type; or if you want to pass arguments to the baseClass's Sub New that are different from those passed to the derived class's Sub New.

In the following script, the derived class's Sub New passes two variables declared at module level to the base class.

```
Class Fruit
  Public weight As Single
  Public color As String
  Sub New(w As Single, c As String)
    weight! = w!
    color$ = c$
    Print "Fruit New() weight = ";w!, "color =";c$
  End Sub
End Class

Class Banana As Fruit
  Sub Peel
  '...
  End Sub

  ' Banana accepts only a weight.  The Sub New passes both
  ' weight and color to the base class (Fruit).
  Sub New(w As Single), Fruit (w, "Yellow")
    '...
    Print "Banana New() Weight = ";w!
  End Sub
End Class

Dim z As New Banana (0.45)   ' Create a .45 lb yellow banana.
```

Calling Sub New and Sub Delete

When LotusScript creates an object of a derived class, the call to the Sub New for the derived class generates a call of the Sub New for the base class. If that base class is itself a derived class, LotusScript calls its base class, and so on. After all the calls, the highest-level Sub New is executed followed by the Sub New of every class in the derivation chain. The Sub New of the class of the object being created is executed last.

When LotusScript deletes an object of a derived class, it calls the Sub Delete for the derived class, followed by the Sub Delete of the base class's Sub Delete, and so on for every class in the derivation chain, up to the highest base class; that is, in the reverse order of the Sub New execution.

The following example demonstrates the order in which Sub New and Sub Delete are called.

```
Class Fruit
  Public weight As Single
  Public color As String

   Sub New(w As Single, c As String)
    weight! = w!
    color$ = c$
    Print "Fruit: New"
   End Sub

  Sub Delete
    Print "Fruit: Delete"
  End Sub
End Class

Class Apple As Fruit
  Public seedCount As Integer

  Sub Core
  ' ...
  End Sub

  Sub New(w As Single, c As String)
    Print "Apple: New"
  End Sub

  Sub Delete
    Print "Apple: Delete"
  End Sub

End Class

Dim y As New Apple(1.14, "Red")
' Executes Fruit's Sub New and then Apple's Sub New.

Delete y
' Executes Apple's Sub Delete and then Fruit's Sub Delete.
```

Accessing base-class properties and methods

A derived class can call a property or method in a base class, even if that method was overridden in the derived class. You use two dots (dotdot notation) to access a base class's overridden method. Dotdot notation is valid only in class scope (within a Class statement).

The syntax to call an overridden property or method is:

baseClassName..*propertyName* (*parameters*)

or

baseClassName..*methodName* (*parameters*)

For example, you can override a method just to add additional processing. You would call the base class's method and then do the extra processing in the derived class method.

Using object references as arguments and return values

You can pass an object reference as an argument to a method, or to any procedure defined to accept it. You can also use an object reference as the return value of a procedure. LotusScript passes objects by reference, not by value.

Keep these rules in mind when you pass an object reference to a procedure:

- You can pass a reference to a derived-class object to a procedure if the procedure parameter is declared as a variable of the base class.
- You cannot pass a reference to a base-class object if the procedure's parameter is declared as a variable of the derived class.

For example, the following script defines the PrintAccount sub at module level to take an object as an argument:

```
Class Account
  Sub DepositMoney
    Print "In Account's DepositMoney sub."
  End Sub
End Class

Class CheckingAccount As Account
  Sub DepositMoney
    Print "In CheckingAccount's DepositMoney sub."
  End Sub
End Class

Sub PrintAccount(AccountArg As Account)
  Call AccountArg.DepositMoney
End Sub

Dim X As New Account
Call PrintAccount(X)      'Calls Account's DepositMoney method.
```

```
Dim Y As New CheckingAccount
' Calls CheckingAccount's DepositMoney sub. Y is legal as an
' argument to PrintAccount, because CheckingAccount is a derived
' class of Account.
Call PrintAccount(Y)
```

Using the Set statement with derived class objects

You can assign a variable that contains a reference to a derived-class object to a variable that can contain a reference to any of that object's base classes. For example, you can assign the value of a variable of type CheckingAccount to a variable of type Account because the CheckingAccount class is derived from the Account class.

You cannot assign a reference in a variable of a base class to a variable that refers to an object of a derived class. For example, you cannot assign a reference in a variable of the Account class to a variable of the CheckingAccount class. If such an assignment were allowed, you might expect to be able to use CheckingAccount's methods on the referenced object. But they might not exist, since the object might be of the Account class.

```
Class Account
  '...
End Class

Class CheckingAccount As Account
  '...
End Class

Dim X As New Account
Dim Y As New Account
Dim Z As New CheckingAccount

' Legal assignment of the contents of a base-class variable
' to a base-class variable
Set X = Y

' Legal assignment of the contents of a derived-class variable
' to a base-class variable
Set X = Z

' Cannot assign base-class variable to derived-class variable.
Set Z = X          ' Illegal
```

The last statement is illegal because, following the Set X = Z statement, the variable X references an object of the derived class, CheckingAccount. But the statement Set Z = X attempts to assign the value of a base class object reference variable, X, to a derived class object reference variable, Z.

Arrays and Lists of Classes

If you're working with groups of objects, you can create an array or list that includes the objects as elements. The following example creates an array of Fruit objects, based on the earlier declaration of the Fruit class:

```
' Declare an array of references to the base class; a Fruit basket.
Dim Basket List As Fruit
Set Basket(1) = New Apple(0.86, "Green", "Macintosh", 24)
Set Basket(2) = New Apple(0.98, "Red", "Delicious",33)
Set Basket(3) = New Banana(0.32, "Yellow")
Set Basket(4) = New Apple(1.2, "Yellow", "Delicious",35)

' Prepare all of the fruit in the basket.
ForAll YummyThing in Basket
  YummyThing.Prepare   ' Call each object's Prepare sub.
End ForAll
```

Chapter 6
Expressions and Operators

An **expression** is a sequence of operators and operands that evaluates to a single value at run time. An **operand** is a language element that represents a value, and an **operator** is a language element that determines how the value of an expression is to be computed from its operand or operands. A **unary operator** performs an operation on a single operand, and a **binary operator** performs an operation on two operands.

An expression can consist of any of the following:

- A literal value—for example, the integer 5 or the string "my cat Geoffrey"
- A constant, variable, property, or function representing a single value—for example, anInteger%, aString$, checkBox1.State, CStr(anInt%)
- One or another of the above plus a unary operator—for example, - anInt%
- Two of the above separated by a binary operator—for example, anInt% * anotherInt%
- Two other expressions separated by a binary operator—for example, (anInt% > 0) And (anInt% <= 10)

All legal expressions evaluate to a numeric value, a String value (possibly the empty string), NULL, EMPTY, NOTHING, or the Boolean value True (-1) or False (0).

The rest of this chapter describes the set of LotusScript operators, how they may be combined with operands to form expressions, and how those expressions are evaluated.

Operators

LotusScript supports the following kinds of operators:

- Arithmetic, for performing basic mathematical operations such as addition, for example:

  ```
  anInt% + anotherInt%.
  ```

- Concatenation, for joining discrete elements to form a single string, for example:

  ```
  "My cat " & "Geoffrey".
  ```

- Relational (comparison), for comparing values, for example:

```
' Test if the value of anInt% is less than or equal to
' the value of anotherInt%.
anInt% <= anotherInt%
```

- Logical (bitwise), for performing bitwise arithmetic, for example:

```
' Calculate the logical product of binary 10 and 11.
2 And 3
```

- Logical (Boolean), for testing expressions for their truth value (True or False), for example:

```
' Test if the value of anInt% is between 1 and 10, inclusive.
(anInt% > 0) And (anInt% <= 10)
```

  Because LotusScript uses bitwise arithmetic to determine the truth value of an expression, the distinction between bitwise and Boolean operators is a somewhat artificial one, however convenient (see “Logical operators” later in this chapter).

- Assignment, for assigning values to variables and properties, for example:

```
 'Add the values of anInt% and anotherInt% and assign the result
' to the variable newInt%.
newInt% = anInt% + anotherInt%
```

- The Is operator for comparing the values of object reference variables to see if they are equal. Its operands may either be object reference variables or the constant NOTHING. The Is operation returns True (-1) if both operands are variables containing a reference to the same object, or if both operands evaluate to NOTHING. Otherwise the Is operation returns False (0). For example:

```
Class ClassA
'...
End Class
Dim X As New ClassA
Dim Y As ClassA
Set Y = X
Print X Is Y
' Output: True
```

The following sections summarize the operators available in LotusScript.

Numeric operators

The table below summarizes the operators you can use in expressions whose operands represent numeric values.

Type of operator	*Operator*	*Operation*
Arithmetic	^	Exponentiation
	-, +	Unary negation (unary minus), unary plus
	*, /	Multiplication, floating-point division
	\	Integer division
	Mod	Modulo division (remainder)
	-, +	Subtraction, addition
Relational (comparison)	=, <>, ><, <, <=, =<, >, >=, =>	Equal, not equal, not equal, less than, less than or equal to, greater than, greater than or equal to, greater than or equal to
Logical (bitwise)	Not	One's complement
	And	Bitwise And
	Or	Bitwise Or
	Xor	Bitwise exclusive Or
	Eqv	Bitwise equivalence
	Imp	Bitwise implication
Logical (Boolean)	Not	Logical negation
	And	Logical And
	Or	Logical Or
	Xo	Logical exclusive Or
	Eqv	Logical equivalence
	Imp	Logical implication

Arithmetic operators

When an arithmetic expression contains a NULL operand, the expression as a whole evaluates to NULL. For example:

```
Dim varV
Dim anInt%
varV = NULL
varV = varV ^ 2
' Test to see if varV is NULL.
Print IsNull (varV)
' Output: True
anInt% = 5
Print IsNull(varV * anInt%)
' Output: True
```

Note that only variables of type Variant may be assigned a value of NULL without causing an error. Thus, the following is perfectly acceptable:

```
varV = NULL
varV = varV * 5
```

but the following is not:

```
anInt% = anInt% * varV
' Generate an error.
```

When the result of an arithmetic operation is too large for the type of variable to which it is assigned, LotusScript performs the necessary data type conversion (if possible):

```
Dim anInt As Integer
Dim aNumericV As Variant
aNumericV = 10000 ^ 10
Print aNumericV
' Output: 1E+40
Print TypeName(aNumericV)
' Output: DOUBLE
anInt% = 10000 ^ 10
' Generate an error.
```

LotusScript also performs the necessary rounding when performing floating point division or modulo arithmetic on integer operands:

```
aDouble# = 42.5
anInt% = 64
anInt% = anInt% / 7
Print anInt%
' Output: 9
Print aDouble# Mod anInt%
' Output: 6
```

For more information on data type conversion and rounding, see "Automatic data type conversion" in Chapter 3.

Relational (comparison) operators

An expression consisting of two numeric operands and a relational (comparison) operator evaluates to True (-1), False (0), or, if either or both of the operands is NULL, to NULL. (For a description of the way in which LotusScript treats the values True (-1) and False (0), see "Boolean values" in Chapter 3, "Data Types, Constants, and Variables"). For example:

```
anInt% = 10
anotherInt% = 15
Dim theResultV As Variant

If anInt% > anotherInt% Then
   Print anInt% & " is greater than " & anotherInt% & "."
Else
   Print anInt% & " is less than or equal to " & anotherInt% & "."
End If
' Output: 10 is less than or equal to 15.
theResultV = (anInt% > anotherInt%)
Print theResultV
' Output: False
Print CInt(anInt% > anotherInt%)
' Output: 0
Print (anInt% > anotherInt%) = False
' Output: True
' because the expression (anInt% > anotherInt%) = False
' is True.
```

When the operands in a comparison are of different data types, LotusScript performs the necessary conversion where possible to make the operands compatible before doing the comparison:

- LotusScript converts an EMPTY-valued operand to 0 if the other operand is numeric.
- When LotusScript performs a comparison operation on operands of different numeric data types, the value of the operand with the lower type is promoted to the higher type before the operation is carried out. The ordering of the numeric data types, from lowest (Integer) to highest (Currency), is as follows:

  Integer

  Long

  Single

  Double

  Currency

Conversion of a value of type Single or Double to a value of type Currency may cause overflow or loss of precision.

When a Single value is compared to a Double, the Double is rounded to the precision of the Single.

- Relational operations on date/time values are performed on both the date and the time. For two date/time values to be equal, both their date and time portions must be equal. For inequality, either the date or time portion may be unequal. For all other operations, the comparison is first done on the date portions. If the date portions are equal, the comparison is then done on the time.

Logical operators

You use the logical operators And, Or, Xor, Eqv, and Imp to perform two kinds of operation, which are functionally different but involve essentially the same underlying mechanism:

- Bitwise, to compare the bits in the binary representation of two numeric values and return a new number derived from that comparison. For example:

```
' Calculate the logical product of binary 10 and 11
' and display the result in binary representation.
Print Bin$(2 And 3)
' Output: 10
```

- Boolean, to test the truth value of a two-operand expression and return a value of True (-1), False (0), or NULL. In a Boolean operation, LotusScript compares the bits in the binary representation of the truth values for each operand and returns a value derived from that comparison. For example:

```
Dim anInt% As Integer
anInt% = 5
Print (anInt% > 2) And (anInt% < 10)  ' Both operands are True.
' Output: True
Print CInt((anInt% > 2) And (anInt% < 10))
' Output: -1
Print CInt(True And True)
' Output: -1
```

You use the logical operator Not to perform the same sorts of operations on expressions consisting of a single operand. Not simply reverses the values of the bits in the binary representation of its operand. For example:

```
Print Bin$(Not 3)
' Output: 11111111 11111111 11111111 11111100

Print Bin$(Not False)
' Output: 11111111 11111111 11111111 11111111
Print (Not True)
' Output: 0
```

Bitwise operators

An expression consisting of the bitwise operator Not and a numeric operand evaluates to an Integer or Long value (or to NULL if the operand is NULL). This number is the result of reversing the values of the bits in the binary representation of the operand (one's complement). For example:

```
anInt% = 8
Print Bin$(anInt%)
' Output: 1000
anotherInt% = Not anInt%
Print Bin$(anotherInt%)
' Output: 11111111 11111111 11111111 11110111
```

An expression consisting of two numeric operands and a bitwise operator evaluates to an Integer or Long value (or to NULL if one of the operands is NULL). The rules that determine the data type of the result of a bitwise operation are the following:

- LotusScript converts an EMPTY-valued operand to 0.
- LotusScript rounds a floating-point operand to an integer using the rules described in "Automatic data type conversion" in Chapter 3. The data type of the operand is Long.
- If an operand is a date/time value, LotusScript uses the numeric value of the date as the operand. The data type of the operand is Long.

The following table summarizes the results of bitwise operations on two-operand expressions:

Operator	*If bit n in expr1 is*	*And bit n in expr2 is*	*Then bit n in the result is*
And	0	0	0
	0	1	0
	1	0	0
	1	1	1
Or	0	0	0
	0	1	1
	1	0	1
	1	1	1
Xor	0	0	0
	0	1	1
	1	0	1
	1	1	0

Continued

Operator	*If bit n in expr1 is*	*And bit n in expr2 is*	*Then bit n in the result is*
Eqv	0	0	1
	0	1	0
	1	0	0
	1	1	1
Imp	0	0	1
	0	1	1
	1	0	0
	1	1	1

The following example illustrates the use of the bitwise operators:

```
anInt% = 10
anotherInt% = 5
aDouble# = 2.6
Print Bin$(anInt%)
' Output: 1010
Print Bin$(anotherInt%)
' Output: 101
Print Bin$(aDouble#)
' Output: 11

theResult% = anInt% And anotherInt%
Print Bin$(theResult%)
' Output: 0
theResult% = anInt% And aDouble#
Print Bin$(theResult%)
' Output: 10

theResult% = anInt% Or anotherInt%
Print Bin$(theResult%)
' Output: 1111
theResult% = anInt% Or aDouble#
Print Bin$(theResult%)
' Output: 1011

theResult% = anInt% Xor anotherInt%
Print Bin$(theResult%)
' Output: 1111
theResult% = anInt% Xor aDouble#
Print Bin$(theResult%)
' Output: 1001
```

```
theResult% = anInt% Eqv anotherInt%
Print Bin$(theResult%)
' Output: 11111111 11111111 11111111 11110000
theResult% = anInt% Eqv aDouble#
Print Bin$(theResult%)
' Output: 11111111 11111111 11111111 11110110

theResult% = anInt% Imp anotherInt%
Print Bin$(theResult%)
' Output: 11111111 11111111 11111111 11110101
theResult% = anInt% Imp aDouble#
Print Bin$(theResult%)
' Output: 11111111 11111111 11111111 11110111
```

Boolean operators

An expression consisting of two operands and a Boolean operator evaluates to True (-1) if the expression is true, and False (0) if it is false, unless one of the operands is NULL. In that case, the result may be NULL or True or False, depending on the operator and the operand. The following table summarizes the various possibilities:

Operator	*If expr1 is*	*And expr2 is*	*The expression evaluates to*
And	True	True	True
	True	False	False
	False	True	False
	False	False	False
Or	True	True	True
	True	False	True
	False	True	True
	False	False	False
Xor	True	True	False
	True	False	True
	False	True	True
	False	False	False
Eqv	True	True	True
	True	False	False
	False	True	False
	False	False	True

Continued

Operator	*If expr1 is*	*And expr2 is*	*The expression evaluates to*
Imp	True	True	True
	True	False	False
	False	True	True
	False	False	True

When an operand in a numeric expression including a Boolean operator is NULL, the whole expression evaluates to NULL except under the following circumstances:

- If the operator is And and the other operand is False, then the expression evaluates to False.
- If the operator is Or and the other operand is True, then the expression evaluates to True.
- If the operator is Imp and the first operand is False, then the expression evaluates to True.
- If the operator is Imp and the second operand is True, then the expression evaluates to True.

The following example illustrates the use of Boolean operators:

```
' Have the user enter two integers between 1 and 10.
' Test to see if the first (num1%) is less than 6 and
' if the second (num2%) is greater than 5. Display a
' message according to the truth value of the two tests.
Dim num1 As Integer
Dim num2 As Integer
num1% = InputBox("Enter an integer between 1 and 10:")
num2% = InputBox("Enter an integer between 1 and 10:")
Print "num1 = " & num1% & " num2 = " & num2%
If num1% <6 And num2% >5 Then
   Print "And:" & num1% & " is less than 6 and " & num2% & _
   " is greater than 5."
End If
If num1% <6 Or num2% >5 Then
   Print "Or:" & num1% & " is less than 6 or " & num2% & _
   " is greater than 5, or both."
End If
If num1% <6 XOr num2% >5 Then
   Print "XOr: " & num1% & " is less than 6 or " & num2% & _
   " is greater than 5, but not both."
End If
If num1% <6 Eqv num2% >5 Then
   Print "Eqv:" & num1% & " is less than 6 and " & num2% & _
    " is greater than 5, or " & num1% & " is greater than 5 and " & _
    num2% & " is less than 6."
End If
```

```
If num1% <6 Imp num2% >5 Then
   Print "Imp:" & num1% & " is less than 6 and " & num2% & _
   " is greater than 5, or " & num1% & _
   " is greater than 5 and " & num2% & _
   " is less than 6, or " & num1% & _
   " is greater than 5 and " & num2% & " is greater than 5."
End If
' Sample Output:
' num1 = 6 num2 = 6
' Or: 6 is less than 6 or 6 is greater than 5, or both.
' XOr: 6 is less than 6 or 6 is greater than 5, but not both.
' Imp: 6 is less than 6 and 6 is greater than 5, or 6 is
' greater than 5 and 6 is less than 6, or 6
' is greater than 5 and 6 is greater than 5.

' num1 = 10 num2 = 1
' Eqv: 10 is less than 6 and 1 is greater than 5, or 10 is greater
' than 5 and 1 is less than 6.
' Imp: 10 is less than 6 and 1 is greater than 5, or 10 is
' greater than 5 and 1 is less than 6, or
' 10 is greater than 5 and 1 is greater than 5.

' num1 = 5 num2 = 9
' And: 5 is less than 6 and 9 is greater than 5.
' Or: 5 is less than 6 or 9 is greater than 5, or both.
' Eqv: 5 is less than 6 and 9 is greater than 5, or 5 is greater than
' 5 and 9 is less than 6.
' Imp: 5 is less than 6 and 9 is greater than 5, or 5 is
' greater than 5 and 9 is less than 6, or
' 5 is greater than 5 and 9 is greater than 5.
```

String operators

The following table summarizes the operators you can use in expressions whose operands represent string values:

Type of operator	*Operator*	*Operation*
Concatenation	&, +	Concatenation.
Relational (Comparison)	=, <>, ><, <, <=, =<, >, >=, =>	Equal to (same as), not equal to (not same as), not equal to (not same as), earlier in the sort order than, earlier in the sort order than or same as, earlier in the sort order than or same as, later in the sort order than, later in the sort order than or same as, later in the sort order than or same as.
	Like	Contains (substring matching with wildcards)

Concatenation operators

LotusScript offers two operators for concatenating strings (or operands that can be interpreted as strings): ampersand (&) and plus (+). Plus (+) is potentially ambiguous, since it can be interpreted as the arithmetical addition operator. LotusScript determines whether to interpret the plus as a concatenation operator or an addition operator on the basis of the operands in the expression in which it appears. Use the ampersand operator to ensure a concatenation operation.

The result of a concatenation of two strings is a string containing the first operand followed immediately by the second operand. If one of the operands is not a string, LotusScript performs the necessary conversion according to the following rules:

- An EMPTY value is converted to the empty string ("").
- NULL is treated as though it were the empty string.
- Numeric operands are converted to their text representation (if the concatenation operator is the ampersand).
- A date/time value is converted to a date/time string.

For example:

```
anInt% = 123
aString$ = "Hello"
anotherString$ = "world"
varV = NULL
Print aString$ & ", " & anInt% & " " & varV & _
    anotherString$ & "."
' Output: Hello, 123 world.
```

Relational (comparison) operators

You use the relational (comparison) operators =, <>, ><, <, <=, =<, >, >=, and =>to ascertain the relative positions of two strings in ASCII sort order. The result of comparing two strings in this way is a value of True (-1), False (0), or NULL (if one of the operands is NULL). Whether the comparison is case-sensitive or case-insensitive depends on the setting of the Option Compare statement in the module in which the comparison takes place. The default setting (Option Compare Case) means that string comparison is case-sensitive: an uppercase character has a different value—occupies a different position in the sort order—than its lowercase counterpart. The sort order is determined by your country and language settings.

An alternative way of making string comparison case-sensitive is with the Option Compare Binary statement: Option Compare Binary specifies that string comparison is case-sensitive, and the sort order is determined by the platform on which your product is running LotusScript.

Use the Option Compare NoCase statement makes string comparison case-insensitive: an uppercase character and its lowercase counterpart share the same position in the sort order.

The following example illustrates the use of relational operators to perform string comparison. In the example, the user enters a character, which is then checked to see if it falls in the range A-Z. If not, the character is checked to see if it falls in the range a-z.

```
Option Compare Binary
Dim theChar As String
theChar$ = InputBox$("Please enter a character:")
If ((theChar$ >= "A") And (theChar$ <= "Z")) Then
   Print "You entered an uppercase character."
ElseIf ((theChar$ >= "a") And (theChar$ <= "z")) Then
   Print "You entered a lowercase character."
Else
   Print "You entered a nonalphabetic character."
End If
```

Like

You use the Like operator to test a string to see if it matches a text pattern that you specify. You can use wildcard characters in the specification of the text pattern. The operation returns a value of True (-1) if the string matches the text pattern, False (0) if the string does not match the pattern, or NULL if either the string or the contents of the text pattern is NULL. The comparison of string to text pattern is either case-sensitive or case-insensitive, depending on the setting of the Option Compare statement in the module in which the comparison takes place (see the preceding section of this chapter).

Use a statement of the following form to test a string to see if it matches a specified text pattern and assign the result of the test to a variable:

result% = SourceString **Like** *TextPattern*

where *SourceString* is an expression that evaluates or can be converted to a string, and *TextPattern*, which must be enclosed in quotation marks, is a sequence of individual ANSI characters and any of the following wildcard characters or collections of characters alone or in combination:

Wildcard	*Matches*
?	Any one character
#	Any one digit from 0 through 9
*****	Any number of characters (zero or more)
[*characters***]**	Any of the characters in the list or character range specified here
[!*characters***]**	Any character not included in the list or character range specified here

To match characters in a list, enclose the characters between square brackets with no spaces or other delimiters between characters (unless you want the space character to be part of the list). For example, [1, 2, 3, A, B, C] represents the characters 1, comma, space, 2, 3, A, B, and C (the redundant occurrences of the space and comma are ignored). But [123ABC] represents the characters 1, 2, 3, A, B, and C (with no space or comma character).

To match characters in a range, separate the lower and upper bounds with a hyphen, as in [1-5]. Always specify the range in ascending sort order (A-Z rather than Z-A). Sort order is determined by the setting of Option Compare. When you specify multiple ranges, you don't have to separate them with anything: for example, [1-5A-C] contains the ranges 1-5 and uppercase A-C.

Use a character list to match the following characters: comma (,), exclamation mark (!), question mark (?), pound sign (#), asterisk (*), and left bracket ([). To match against a right bracket (]) or hyphen (-), include the character outside any range or list in *TextPattern*.

The following example illustrates the various ways you can test a string with Like to see if it contains a given substring:

```
' Make string comparison case-sensitive.
Option Compare Binary
Dim anArray(1 To 6) As String
anArray(1) = "Juan"
anArray(2) = "Joan"
anArray(3) = "Alejandro"
anArray(4) = "Jonathan"
anArray(5) = "Andrea"
anArray(6) = "Jane"
UB% = UBound(anArray)

' Test each name in anArray$ to see if it contains a substring
' consisting of any characters followed by uppercase J
' followed by any characters followed by lowercase n followed
' by any characters.
For counter% = 1 to UB%
  If anArray(counter%) Like "*J*n*" Then
      Print anArray(counter%) & " " ;
  End If
Next
Print ""
' Output: Juan Joan Jonathan Jane
```

```
' Test the lowercase representation of each name in anArray$
' to see if it contains a substring consisting of any
' characters followed by lowercase j followed by any single
' character followed by lowercase n followed by any characters.
For counter% = 1 to UB%
  If LCase$(anArray(counter%)) Like "*j?n*" Then
      Print anArray(counter%) & " " ;
  End If
Next
Print ""
' Output: Alejandro Jonathan Jane

' Test each name in anArray$ to see if it contains
' a numeric character.
badRec% = 0
For counter% = 1 to UB%
  If anArray(counter%) Like "*#*" Then
      Print anArray(counter%) & " contains a numeral."
      badRec% = badRec% + 1
  End If
Next
If badRec% = 0 Then
  Print "No name contains a numeral."
End If
' Output: No name contains a numeral.

' Test the lowercase representation of each name in anArray$
' to see if it ends in a vowel.
For counter% = 1 to UB%
  If anArray(counter%) Like "*[aeiou]" Then
      Print anArray(counter%) & " " ;
  End If
Next
Print ""
' Output: Alejandro Andrea Jane

' Test each name in anArray$ to see if it begins with a
' character in the range of uppercase A to uppercase C,
' inclusive.
For counter% = 1 to UB%
  If anArray(counter%) Like "[A-C]*" Then
      Print anArray(counter%) & " " ;
  End If
Next
Print ""
' Output: Alejandro Andrea
```

```
' Test each name in anArray$ to see if it begins with a
' character other than uppercase J.
For counter% = 1 to UB%
  If anArray(counter%) Like "[!J]*" Then
      Print anArray(counter%) & " " ;
  End If
Next
Print ""
' Output: Alejandro Andrea

' LotusScript converts a numeric value to a string
' when it is the operand to the left of the Like operator.
anInt% = 12345
aDouble# = 123.45
' See if anInt% starts with two digits followed
' by the digits 23 followed by zero or more digits (or characters).
If anInt% Like "##34*" Then
  Print "We have a match."
End If
' Output: We have a match.
' See if aDouble# starts with two digits followed by a 3
' followed by a single character (presumably a period or a comma)
' followed by any digit followed by zero or more digits (or
characters).
If aDouble# Like "##3?#*" Then
  Print "We have a match."
End If
' Output: We have a match.
```

Precedence and associativity

Rules of precedence and associativity determine the way an expression with multiple operands is evaluated. **Rules of precedence** determine the order in which different types of operations are performed, and **rules of associativity** determine the order in which operations of equal precedence are performed. In general terms, arithmetic operations are performed first, then comparison operations, then logical operations, and then assignment. The general rule governing associativity is that it proceeds from left to right.

Within each general type of operation, there are also rules of precedence and associativity that determine the order in which operations are performed. In order of highest-to-lowest, the precedence of LotusScript operators is as follows:

Type of operator	*Operator*	*Operation*
Arithmetic	^	Exponentiation
	-	Unary negation (unary minus)
	*, /	Multiplication, floating-point division
	\	Integer division
	Mod	Modulo division (remainder)
	-, +	Subtraction, addition
Concatenation	&	String concatenation
Relational (Comparison)	=, <>, ><, <, <=, =<, >, >=, =>	Numeric comparison Equal to, not equal, to, not equal to, less than, less than or equal to, less than or equal to, greater than, greater than or equal to, greater than or equal to String comparison: Equal to, not equal to, not equal to, less than, less than or equal to, less than or equal to, greater than, greater than or equal to, greater than or equal to
	Like	Contains (substring matching)
Logical	Not	Logical negation or one's complement
	And	Boolean or bitwise And
	Or	Boolean or bitwise Or
	Xor	Boolean or bitwise exclusive Or
	Eqv	Boolean or bitwise logical equivalence
	Imp	Boolean or bitwise logical implication
Object reference comparison	Is	Refers to the same object
Assignment	=	Assignment

Associativity rules govern the order in which operators of equal precedence are evaluated: binary operations of equal precedence are performed left to right.

The following statement illustrates the way precedence and associativity work. The operations of multiplication and division are performed first in left to right order; then the subtraction operation; and then the bitwise And operation.

```
Print 4 And 10 - 2 * 3 / 2
' Output: 4 because 2 * 3 = 6
'         6 / 2 = 3
'         10 - 3 = 7 (binary 111)
'         4 (binary 100) And 7 (binary 111) = 4 (binary 100).
```

You can alter the default order in which operations are performed by enclosing the expressions you want evaluated first in parentheses. For example:

```
anInt% = 5
anotherInt% = 10
aThirdInt% = 7
print anInt% - (anotherInt% + aThirdInt%)
' Output: -12
```

or, alternatively:

```
theResult% = -1 Or -1 Imp 0
Print theResult%
' Output: False
' because -1 Or -1 = True, and True Imp 0 is False.
theResult% = -1 Or (-1 Imp 0)
Print theResult%
' Output: True
' because -1 Imp 0 is False, and -1 Or False is True.
```

Note A function is evaluated before any of the operators in an expression. For example:

```
Print -1 > 0
' Output: False
Print Abs(-1) > 0
' Output: True
```

Chapter 7
Directing Traffic Within an Application

The flow of execution of a script generally follows the sequence of statements in the script. However, certain statements and conditions alter the flow of execution. These are summarized in the following section of this chapter, “Flow of Execution.”

The remainder of this chapter, the section “Flow Control Statements,” describes the behavior of particular statements that alter the flow of execution: block statements, branching statements, and the End and Exit statements. (For definitions of these kinds of statements, see the following section, “Flow of Execution.”) The flow of execution may also be changed at run time by the occurrence of an error. For information on the statements for processing run-time errors in a script, see Chapter 8, “Error Processing.”

Flow of Execution

Comments are not executed at all. These include any source text preceded on a line by the comment marker apostrophe ('), the text in a Rem statement, and the text enclosed between the compiler directives %Rem and %End Rem. The LotusScript compiler reads and discards these.

The **compiler directive** %Include directs the compiler to replace the directive by other text before continuing to compile. The compiler directive %If directs the compiler to select or omit text contained within the scope of the directive, replacing the directive by the selected text. The result of the replacement based on %Include or %If is compiled as if it appeared in the original script. The flow of execution in the compiled result follows the same rules as the flow of execution in the rest of the script.

Declarations include the Declare statement for forward references, the Declare statement for external C calls, the Const statement, and the Dim statement. With one exception, declarations do not product executable code. The result of a declaration is information about a procedure, a variable, or a constant; for example, its type, dimensions, or value. This governs the behavior of the script that uses the declared item; but the declaration itself is not executed when the script runs. The exception is a Dim statement that includes the keyword New. The result of this statement is to construct a new object of a class; and this is done when the script is executed. This is the only declaration that generates executable code.

A few other statements also produce no executable code. These include Option Base, Option Compare, Option Declare, and Option Public; the Type statement; and the Def*type* statements.

Besides the Type statement, the **definition statements** include the Class statement and the procedure definition statements: Function, Sub, Get Property, and Set Property. While these definition statements produce executable code, this code is not executed in place. LotusScript executes a procedure only when it is explicitly invoked. When the procedure completes execution, the script execution continues from the point where the procedure was invoked. There are two pairs of procedures, however, that are executed without being explicitly invoked:

- Sub New and Sub Delete

  These are executed when an object is created or deleted, respectively.

- Sub Initialize and Sub Terminate

  Sub Initialize is executed when the compiled module representing the script is loaded. Sub Terminate is executed when the module is unloaded.

Certain **block statements** define an altered flow of execution within the body of the statement. These include If...Then...Else, If...Then...ElseIf, and Select Case; and the **iterative block statements** Do, While, For, and ForAll.

Branching statements also specify an altered flow of execution. These are GoTo, If...GoTo...Else, On...GoTo, GoSub, On...GoSub, and Return. When one of these is executed, the flow of execution depends in general on run-time conditions.

The End statement and the Exit statement also change the flow of execution:

- The End statement can appear anywhere within a procedure. When the statement is executed, script execution ends.
- The Exit statement can appear within a Do statement, a For statement, or a ForAll statement; or within a procedure. When the Exit statement is executed, LotusScript ends execution of the statement or the procedure. Execution continues as it would following an ordinary completion of the Do, For, or ForAll statement, or following an ordinary return from the procedure.

The flow of execution may also be changed at run time by the occurrence of an error. Either execution ends, or an On Error statement in the script specifies how to respond to the error, in one of these ways:

- By continuing execution with the statement following the statement that caused the error
- By invoking an error handling routine in the current procedure
- By seeking an error handling routine in a procedure within the chain of procedure calls that invoked the current procedure

An error handling routine ends with a Resume statement that directs LotusScript to resume execution either at a designated labeled statement, or at the statement that caused the error, or at the statement following the statement that caused the error.

Note that statement labels can appear only within procedures. A statement at module level in a script — not contained within a procedure — cannot be labeled. Since any given label is known only within the procedure where it is defined, a branching statement that may transfer control to a labeled statement can appear only within the same procedure as the labeled statement. The statements that may transfer control to a labeled statement are GoTo, GoSub, On...GoTo, On...GoSub, If...GoTo...Else, and Resume. If an error occurs that is governed by an On Error...GoTo *label* statement, the On Error statement and the labeled statement must be in the same procedure.

The block statements, the branching statements, and the End and Exit statements are collectively called **flow control statements**. The rest of this chapter describes these statements.

Flow Control Statements

The flow control statements fall into several functional groups:

- The block statements that specify executing one or another group of subsidiary statements, depending on specified conditions. These statements are If...Then...Else, If...Then...ElseIf, and Select Case.
- Branching statements, which specify continuing execution at some other point in the script, possibly depending on specified conditions. These statements are GoTo, If...GoTo...Else, On...GoTo, GoSub, On...GoSub, and Return.
- Early termination statements (Exit and End), which specify returning from a procedure, or ending execution of a Do, For, or ForAll statement, before execution reaches the statement that ends the procedure or the statement.
- The iterative block statements, which specify repeating a group of subsidiary statements some number of times, or while or until some specified condition is satisfied. These statements are Do, While, For, and ForAll.

The remaining sections in this chapter discuss these statements in the order listed above.

The following are general comments that apply to some or all of the flow control statements.

- The flow of control within each of these statements is different for each statement.
- The flow of control when the statement has completed executing is as follows:
  - The branching statements direct LotusScript to continue executing the script elsewhere, by jumping from the current location to another location.

- The End statement terminates execution of the current procedure, and also execution of any procedure in the sequence of calls that called the current one. Exit Sub, Exit Function, and Exit Property cause an immediate return from the procedure. Exit Do, Exit For, and Exit ForAll cause execution to continue at the statement following the end of the Do, For, or ForAll statement.
- The other flow control statements cause execution to continue at the statement following the end of the flow control statement, unless a branching statement or an Exit or End statement within the flow control statement causes a jump before the end is reached. The branching statements If...GoTo...Else, On...GoTo, and On...GoSub behave this way also if the condition that triggers the jump is not true.

- There is no built-in limit on the level of nesting of these statements. For example, a Do statement may contain another Do statement that contains a third Do statement, and so on.
- Flow control statements may be nested within each other. For example, a Do statement may contain a For statement that contains another Do statement.

If...Then...Else statement

The If...Then...Else statement specifies conditional execution of either one group or another group of statements, depending on the value of an expression. Each statement group is usually just one short statement, since the entire If...Then...Else statement must be written on one line.

You commonly use the If...Then...Else statement in one of these two forms:

- **If** *condition* **Then** *statements* **Else** *statements*

  In this form, either the Then clause is executed (if *condition* is TRUE); or the Else clause is executed (if *condition* is FALSE). For example:

  ```
  If doCount% >= 1000 Then flagForm% = -1 Else flagForm% = 0
  ```

- **If** *condition* **Then** *statements*

  In this form, the Then clause is executed if *condition* is TRUE; otherwise, nothing is executed. For example:

  ```
  If doCount% >= 1000 Then flagForm% = -1
  ```

For either form, execution continues with the statement on the next line. Nothing can follow the If...Then...Else statement on the same line, since LotusScript recognizes a newline as the If...Then...Else statement terminator.

In the following example, the Then clause consists of the single statement Exit Do; and there is no Else clause.

```
' This example computes the elapsed time to execute
' 1000 iterations of a simple Do loop.
' Time may vary, depending on the workstation.
Dim doCount As Integer, startTime As Single
startTime! = Timer()
doCount% = 0

Do
   ' Increment doCount% through 1000 iterations of the Do loop.
   doCount% = doCount% + 1
   If doCount% > 1000 Then Exit Do
Loop
' Come here upon exit from the Do loop.
Print Timer() - startTime! "seconds for 1000 iterations"
' Output:
' .109375 seconds for 1000 iterations
```

For more information about the Do and Exit statements, see the sections on these statements later in this chapter.

To include more than one statement in the Then clause, separate the statements by the colon (:) statement separator, as in this variation:

```
Do
   If doCount% >= 1000 Then Print "Done." : Exit Do
Loop
```

You can rewrite the two statements in the Do loop in the preceding example as a single If...Then...Else statement. The Do loop then looks like this:

```
Do
   If doCount% >= 1000 Then Exit Do Else doCount% = doCount% + 1
Loop
```

This is a more compact loop than the one in the preceding example, but it runs more slowly.

The condition in the If...Then...Else statement can be simple, as in the preceding example, or complex. Here is an If...Then statement with a more complex condition:

```
If Abs(tempProx! - approx!) >= .00001 And iters% < 40 Then Exit Do
```

LotusScript identifies a statement as an If...Then...Else statement provided it has the form If *condition* Then, or If *condition* Then *statements* Else, followed on the same line by more source code. Unless this language appears on the same line, LotusScript interprets the statement as an If...Then...ElseIf statement.

You can extend the statement to more than one line, by ending each line except the last with the line-continuation character, an underscore (_). But if the statement is long enough to force continuation onto a second line, it may be more readable to rewrite it as an If...Then...ElseIf statement. (For more information, see the next section.)

If...Then...ElseIf statement

The If...Then...ElseIf statement specifies conditional execution of one or another group of statements, depending on whether one or more expressions evaluates to TRUE or FALSE. The syntax is:

If *condition* **Then**

statements

[**ElseIf** *condition* **Then**

statements]

[**ElseIf** *condition* **Then**

statements]

...

[**Else**

statements]

End If

The line breaks in actual statements must appear just as shown in the syntax diagram.

Only one group of statements is executed: either the group following the first condition that evaluates to TRUE, or else those statements following the Else keyword. (If no condition evaluates to TRUE and there is no Else clause, then no statements are executed.) Once a group of statements is executed, no further condition expressions are evaluated; so the order of the ElseIf clauses is important. Program execution continues with the first statement following the End If keywords.

The following example using If...Then...ElseIf statements demonstrates making a user-supplied whole number into an ordinal by adding the appropriate English suffix, such as “st” for 1 and “th” for 17. The script responds differently to numbers outside the range 0 to 50 (an arbitrary limit) and to numbers with a fractional part. In the script, note the nesting of three levels of If...Then...ElseIf statement. The example illustrates that each statement needs its own End If phrase. An End If phrase ends only the innermost statement that hasn’t yet ended.

```
Dim anInt As String, lastDigit As String, printNum As String
anInt$ = InputBox$("Enter a whole number between 0 and 50:")
' Test for a number; print message if not, and do nothing more.
If Not IsNumeric(anInt$) Then
   MessageBox("That's not a number.")
' Test for whole number; print message if not, and do nothing more.
ElseIf Fraction(CSng(anInt$)) <> 0 Then
   MessageBox("That's not a whole number.")
Else
   ' Test for number within required range.
   If CInt(anInt$) <= 50 And CInt(anInt$) >= 0 Then
      ' Number is within range. Find and append the correct suffix.
      lastDigit$ = Right$(anInt$, 1)
      If lastDigit$ = "1" And anInt$ <> "11" Then
         printNum$ = anInt$ & "st"
      ElseIf lastDigit$ = "2" And anInt$ <> "12" Then
         printNum$ = anInt$ & "nd"
      ElseIf lastDigit$ = "3" And anInt$ <> "13" Then
         printNum$ = anInt$ & "rd"
      Else
         printNum$ = anInt$ & "th"
      End If
       ' Print the ordinal in a message box.
      MessageBox("This is the " & printNum$ & " number.")
   Else
      ' Number is out of range. Print message, and do nothing more.
      MessageBox("That number's out of range.")
   End If
End If
' Output:
' (For user input 3): "This is the 3rd number."
' (For user input -5.1): "That's not a whole number."
' (For user input 51): "That number's out of range."
' (For user input abacus): "That's not a number."
```

The example would be easier to read if the conditional processing were not nested three levels deep. However, an If...Then...ElseIf statement that is not included within another statement can be skipped during execution only by executing a transfer of control: either by an Exit or End statement or by a transfer to a labeled statement. All of these mechanisms — the Exit statement, the End statement, GoTo, GoSub, and labels — are illegal outside of a procedure. Thus, if the main logic of this script were made into the contents of a procedure, it could be rewritten more simply. For this algorithm, that would be the conventional way of doing it. Later in this chapter, the same task is performed in a Select Case statement.

The contents of the If clause, the ElseIf clauses, and the Else clause must be written in the correct order. In the following example, look at the order of the contents of the If and ElseIf and Else clauses:

```
Dim timeTest As Single
timeTest! = Timer()   ' The Timer function returns
                      ' the number of seconds elapsed since midnight.
If timeTest! < 43200 Then
   Print "Morning"
ElseIf timeTest! < 64800 Then
   Print "Afternoon"
Else
   Print "Evening"
End If
```

The following example shows the result of exchanging the order of the contents of the If clause and the ElseIf clause. Executing the example with this different order yields a wrong result for a Timer() value of 38017, for example. The Timer() value represents a mid-morning time, but the example prints Afternoon.

```
Dim timeTest As Single
timeTest! = Timer()   ' The Timer function returns
                      ' the number of seconds elapsed since midnight.
If timeTest! < 64800 Then
   Print "Afternoon"
ElseIf timeTest! < 43200 Then
   Print "Morning"
Else
   Print "Evening"
End If
```

Select Case statement

The Select Case statement specifies conditional execution of one group of statements selected from one or more groups, depending on the value of an expression. It is similar to the If...Then...ElseIf statement.

The syntax is:

Select Case *selectExpr*

[**Case** *conditionList*

[*statements*]]

[**Case** *conditionList*

[*statements*]]

...

[**Case Else**

[*statements*]]

End Select

At run time, the Select Case statement compares the value of a single *selectExpr* expression with the values established by each *conditionList*. It executes the *statements* for the first *conditionList* matched by the value of *selectExpr*. Either a single group of statements is executed, or none is executed. If you include a Case Else clause, it's executed only if *selectExpr* fails all conditions in all condition lists. After a clause is executed, LotusScript continues execution at the first statement following the End Select statement.

The following example of Select Case replicates the main example used to illustrate If...Then...ElseIf, which adds a suffix to a whole number to turn it into an ordinal number. The results of the two examples are identical for all user inputs. In this version, however, the principal logic is written as the body of a function definition rather than as module-level statements, and the function is called by a module-level statement. The function accepts a string and returns one of the strings shown. The function is called with the string obtained through InputBox$; after the function returns, the MessageBox statement writes the output.

```
' This script defines and calls the function SetOrd.
' This function accepts a string argument, determines whether it is
' of the right kind, and returns either a message about the argument,
' or a string showing the argument with the correct suffix.
Function SetOrd (anInt As String) As String
   Dim printNum As String
   ' If argument can't be converted to a number,
   ' assign a message and do nothing more.
   If Not IsNumeric(anInt$) Then
      SetOrd$ = "That's not a number."
      Exit Function
   ' If argument is not a whole number,
   ' assign a message and do nothing more.
   ElseIf Fraction(CSng(anInt$)) <> 0 Then
      SetOrd$ = "That's not a whole number."
      Exit Function
```

```
   ' If number is not in range, assign a message and do nothing more.
   ElseIf CInt(anInt$) > 50 Or CInt(anInt$) < 0 Then
      SetOrd$ = "That number's out of range."
      Exit Function
   End If
   ' Determine and append the correct suffix.
   Select Case anInt$
      Case "1", "21", "31", "41": printNum$ = anInt$ & "st"
      Case "2", "22", "32", "42": printNum$ = anInt$ & "nd"
      Case "3", "23", "33", "43": printNum$ = anInt$ & "rd"
      Case Else: printNum$ = anInt$ & "th"
   End Select
   SetOrd$ = "This is the " & printNum$ & " number."
End Function
' Call the function.
MessageBox(SetOrd(InputBox$("Enter a whole number between 0 and
50:")))
```

Note the last line of the example. It is the only executable code outside of the function SetOrd in the example. This line instructs the MessageBox statement to display a message based on the user input received by the InputBox$ function. The value entered by the user is passed to SetOrd, which determines what MessageBox displays.

In several ways, the function SetOrd using Select Case is a cleaner, more legible algorithm than the nested If...Then...ElseIf statements of the earlier version:

- There are no nested statements.
- The phrases Case, Select Case, and Case Else clearly mark the set of logical cases.
- Within each Case clause, the condition list enumerates the exact inputs that satisfy the clause. The Right$ function to obtain the last character in a string isn't needed, and there are no comparison operators. The inputs 11, 12, and 13 aren't specified as exceptions.
- On each line beginning with Case, a colon (:) separates the condition from its consequent clause, and each condition and its clause appear on a single line.
- Using the first two Exit Function clauses within the function definition enables the function to return control to the caller at either of two points without executing the rest of the function.

Also, the algorithm now reflects the logical separation of operations:

- The function performs the manipulations on the number entered by the user.
- Only the input and output operations and the function call run outside the procedure.

GoTo and If...GoTo...Else statements

The GoTo statement transfers control unconditionally. The syntax is:

GoTo *label*

When this statement is executed, LotusScript transfers control to the statement labeled *label*. The location of the GoTo statement in the procedure is unrelated to the location of a labeled statement that it transfers control to. The only requirement is that the GoTo and its target labeled statement must be in the same procedure. The actual flow of control is determined at run time.

The following example uses a GoTo statement to transfer control appropriately within a sub that checks how closely a number approximates PI. A user types a guess at the value of PI to some number of digits, and the script checks the value and reports on it.

```
Sub ApproxPi(partPi As Double)
   Dim reportMsg As String
   ' See how good the approximation is, and assign a response message.
   reportMsg$ = "Not close at all"
   If Abs(PI - partPi#) < 1E-12 Then
      reportMsg$ = "Very close"
      GoTo MsgDone
   End If
   If Abs(PI - partPi#) < 1E-6 Then reportMsg$ = "Close but not very"
   ' Print the message and leave.
MsgDone: MessageBox(reportMsg$)
End Sub
' Ask the user to guess at PI; then call ApproxPi, and report.
Call ApproxPi(CDbl(InputBox$("A piece of PI, please:")))
```

The effect of the transfer using GoTo in the example is to skip the If statement that checks whether the supplied approximation is “Close but not very.” If it's already known to be “Very close,” it makes no sense to check further.

The following example uses GoTo to iterate through the sequence of calculations .25 ^ .25, .25 ^ (.25 ^ .25), .25 ^ (.25 ^ (.25 ^ .25)), and so on, until either two successive expressions in this sequence are within .0001 of each other, or 40 expressions have been calculated.

```
Sub PowerSeq
   Dim approx As Single, tempProx As Single, iters As Integer
   approx! = .25
   iters% = 1
ReIter:
   tempProx! = approx!
   approx! = .25 ^ tempProx!
   If Abs(tempProx! - approx!) >= .0001 And iters% < 40 Then
      ' Iterate again.
      iters% = iters% + 1
      GoTo ReIter
   End If
   Print approx!, Abs(approx! - tempProx!), "Iterations:" iters%
End Sub
Call PowerSeq()
' Output:
' .5000286     6.973743E-05     Iterations: 25
```

In this example, GoTo initiates another iteration. It appears within the Then clause of an If...Then...ElseIf statement. If either two successive expressions are very close in size, or the limit of 40 iterations has been reached, the Then clause is skipped and the next statement following End If, the Print statement, is executed. Then the sub ends.

The example can be generalized to calculate the sequence of values x ^ x, x ^ (x ^ x), and so on, for any value x between 0 and 1, instead of .25 ^ .25, .25 ^ (.25 ^ .25), and so on.

If...GoTo...Else Statement

The If...GoTo...Else statement is simply a convenient way to abbreviate a statement that would otherwise be written If...Then GoTo *label* Else. It can be used when the only action you want to take in the Then clause of an If...Then...Else statement is to transfer unconditionally. The description of If...Then...Else earlier in this chapter applies to this statement, with the GoTo clause substituted for the Then clause. The statement must be written on one line.

For example, here is the executable part of the sub from the preceding example, revised to use If...GoTo (there is no Else clause in this case):

```
   approx! = .25
   iters% = 0
ReIter:
   iters% = iters% + 1
   tempProx! = approx!
   approx! = .25 ^ tempProx!
   If Abs(tempProx! - approx!) >= .0001 And iters% < 40 GoTo ReIter
   Print approx!, Abs(approx! - tempProx!), "Iterations:" iters%
```

This is a more direct expression of the logic of the calculation.

On...GoTo statement

The On...GoTo statement has this syntax:

On *expression* **GoTo** *label*, [, *label*]...

It transfers control to a target label depending on the value of *expression*. It transfers control to the first *label* if *expression* is 1, to the second *label* if *expression* is 2, and so on.

The location of the On...GoTo statement in the procedure is unrelated to the location of a labeled statement that it transfers control to. The only requirement is that the On...GoTo and its target labeled statements must be in the same procedure. The actual flow of control is determined at run time.

The following sub uses On...GoTo to run one of two simple LotusScript performance tests.

```
' By typing 1 or 2 into an input box, the user chooses
' whether to time 1000 iterations of a Do loop,
' or count the number of Yield statements executed in one second.
' Using On...GoTo, the script branches to run one test or the other
' and print the result.
Sub RunPerfTest
   Dim directTempV As Variant, directTest As Integer, i As Integer
   Dim startTime As Single
SpecTest: directTempV = InputBox$(|Type 1 for iteration time,
             or 2 for # of yields:|)
   If Not IsNumeric(directTempV) Then Beep : GoTo SpecTest
   directTest% = CInt(directTempV)
   If directTest% < 1 Or directTest% > 2 _
      Then Beep : GoTo SpecTest
   i% = 0
   ' Branch on 1 or 2.
   On directTest% GoTo TimeCheck, ItersCheck
TimeCheck: startTime! = Timer()
   Do While i% <= 1000
      i% = i% + 1
   Loop
   Print "Time in seconds for 1000 iterations: " Timer() - startTime!
   Exit Sub
ItersCheck: startTime! = Timer()
   Do
      Yield
      i% = i% + 1
   Loop While Timer() < startTime! + 1
   Print "Number of Yields in 1 second: " i%
End Sub
Call RunPerfTest()
```

Three runs of the sub RunPerfTest might have these results, depending on the speed of the computer where LotusScript is running::

```
' Output:
' (With input 2)  Number of Yields in 1 second:  975
' (With input 1)  Time in seconds for 1000 iterations:  .109375
' (With input 2)  Number of Yields in 1 second:  952
```

GoSub, On...GoSub, and Return statements

These three statements have the forms:

GoSub *label*

On *expression* **GoSub** *label* [, *label*]...

Return

When LotusScript encounters a statement of the form GoSub *label*, the following occurs:

- It transfers control to the statement labeled *label*
- It executes statements beginning at *label*, continuing until one of the following occurs:
  - A Return statement is encountered

    In this case, control returns to the statement following the GoSub statement
  - An End statement is encountered; or an Exit Function, Exit Sub, or Exit Property statement is encountered; or an End Function, End Sub, or End Property statement is encountered.

    In these cases, execution of the script ends (End statement), or execution of the enclosing procedure ends (one of the other statements).

Ordinarily, you use the Return statement. In that case, the group of statements executed after the labeled statement and before the Return statement, including any other transfers of control, acts as a subroutine within the current procedure.

The statement On *expression* GoSub *label*, *label*, ... enables transferring control similarly, except that the target label is conditioned on the value of *expression*: control transfers to the first *label* if *expression* is 1, to the second *label* if *expression* is 2, and so on. (Any of these labels may be the same.) The Return statement returns control to the statement following On...GoSub. The next example illustrates this.

The location of the GoSub statement in the procedure is unrelated to the location of a labeled statement that it transfers control to. The only requirement is that the GoSub and its target labeled statements must be in the same procedure. The actual flow of control is determined at run time.

Execution of a GoSub or an On...GoSub statement defines a point of return. Another GoSub or On...GoSub may be executed before a Return statement is executed. When a Return is executed, control returns to the most recently defined point of return. Then that point of return becomes undefined.

Note that the Return statement doesn't return from the procedure. It is a run-time error to attempt to execute a Return statement when there is no currently available point of return within the procedure.

Note also that these statements differ from the GoTo and On...GoTo statements, which transfer control without establishing a point of return.

GoSub and On...GoSub are nonstandard programming techniques with limited usefulness. They enable running a group of statements by transferring control from any number of other locations within the same procedure. Functionally the statements behave as a subroutine, but the pseudo-subroutine is very limited: it can't take arguments, doesn't establish a separate scope, and isn't available from other procedures, or other scripts. It is more common and useful to write the statements as an ordinary sub.

The following example using On...GoSub runs one or the other of two simple performance tests on pieces of the LotusScript language.

```
' By typing 1 or 2 into an input box, the user chooses
' whether to time 1000 iterations of a Do loop,
' or to count the number of Yields executed within one second.
' Using On...GoSub, the script branches to run one test or the other.
' A single Print statement reports the result.
Sub RunPerfTest
   Dim directTempV As Variant, directTest As Integer, i As Integer
   Dim startTime As Single, measure As Single, idPace As String
   SpecTest: directTempV = InputBox$(|Type 1 for iteration time,
               or 2 for # of yields:|)
   If Not IsNumeric(directTempV) Then Exit Sub
   directTest% = CInt(directTempV)
   If directTest% < 1 Or directTest% > 2 Then Beep : GoTo SpecTest
   i% = 0
   ' Branch on 1 or 2.
   On directTest% GoSub TimeCheck, ItersCheck
   ' Return here to print the performance-test result, and leave.
   Print idPace$ measure!
   Exit Sub
```

```
TimeCheck:
   startTime! = Timer()
   Do While i% <= 1000
      i% = i% + 1
   Loop
   measure! = Timer() - startTime!
   idPace$ = "Time in seconds for 1000 Do iterations: "
   Return
ItersCheck:
   startTime! = Timer()
   Do While Timer() < startTime! + 1
      Yield
      i% = i% + 1
   Loop
   measure! = i%
   idPace$ = "Number of Yields in 1 second: "
   Return
End Sub
Call RunPerfTest()
```

Exit statement

The Exit statement terminates execution of a procedure, or a Do, For, or ForAll statement, before execution reaches the end of the procedure definition or the end of the block statement.

The syntax is:

Exit *exitType*

exitType must be one of the keywords Do, For, ForAll, Function, Sub, or Property.

This illustration of Exit appeared already in the discussion of the If...Then...Else statement earlier in the chapter:

```
' Compute the elapsed time to execute 1000 iterations
' of a simple Do loop.
' Time may vary, depending on the workstation.
Dim doCount As Integer, startTime As Single
startTime! = Timer()
doCount% = 0
Do
   ' Increment doCount% through 1000 iterations of the Do loop.
   doCount% = doCount% + 1
   If doCount% > 1000 Then Exit Do
Loop
' Come here upon exit from the Do loop.
Print Timer() - startTime! "seconds for 1000 iterations"
' Output:
' .109375 seconds for 1000 iterations
```

When you use Exit with a Do, For, or ForAll statement, execution continues at the first statement following the end of the block statement.

When you use Exit with a procedure, execution continues as it would following a normal return from the procedure. The following example incorporates the Do statement from the preceding example within a sub:

```
' Compute the elapsed time to execute a sub that runs
' 1000 iterations of a simple Do loop.
Public startTime As Single
Sub ElapsedTime
   Dim doCount As Integer
   doCount% = 0
   Do
      doCount% = doCount% + 1
      If doCount% >= 1000 Then Exit Sub
   Loop
' Because of the Exit Sub statement above, this Print statement
' will not be reached.
Print Timer() - startTime!, "seconds to run 1000 iterations"
End Sub
startTime! = Timer()
Call ElapsedTime()
Print Timer() - startTime! |seconds for sub call to run 1000
iterations|
' Output:
' .109375 seconds for sub call to run 1000 iterations
```

In this example, the Exit Sub statement terminates execution of the sub ElapsedTime after doCount% reaches 1000. Execution continues with the Print statement following the sub call. It is not necessary to terminate execution of the Do loop separately. The Exit Sub statement transfers control from the Do loop out of the sub.

When execution continues after an Exit For statement has run, the count variable for the For statement has its most recent value, just as when execution continues after an ordinary termination of the For statement. When execution continues after an Exit ForAll statement has run, the ForAll alias variable is undefined, just as when execution continues after an ordinary termination of the ForAll statement.

Following execution of an Exit Function statement, the function returns a value to the caller. As with a normal return, this is the last value assigned before the exit. If none was assigned, the function return value is its initialized value: either 0, EMPTY, the empty string (""), or NOTHING. For example:

```
Function TwoVerge(seqSeed As Integer) As Single
   ' Leave if the call argument is not a positive integer.
   ' The return value of TwoVerge is its initial value, 0.
   If seqSeed% < 1 Then Exit Function
   TwoVerge! = Sqr(seqSeed% + 1)
   Dim i As Integer
   For i% = 1 To seqSeed%
      ' TwoVerge computes and returns a value that must be
      ' 1 or greater, according to the following formula.
      TwoVerge! = Sqr(1 + (seqSeed% + 1 - i%) * TwoVerge!)
   Next i%
End Function
```

Here are calls to TwoVerge within Print statements that show the results:

```
Print "Seed:", -1, "Value:" TwoVerge(-1)
Print "Seed:", 20, "Value:" TwoVerge(20)
' Output:
' Seed: -1    Value: 0
' Seed: 20    Value: 1.999998
```

End statement

The End statement terminates execution of the current procedure, and also execution of any procedure in the sequence of calls that called the current one. The syntax is:

End [*returnCode*]

The optional *returnCode* is an integer expression. The script where this statement appears returns the value of this expression to the Lotus product that executed the script. Refer to the product documentation to determine whether the product expects a return value when the End statement is executed. If no return code is expected, do not specify one with the End statement.

The following example is a variation on the script in the preceding section that defined and called the sub ElapsedTime.

```
' Compute the time to run some number of iterations of a For loop,
' and the time to execute the ElapsedTime sub.
Dim anInt As String
Public startSub As Single, startLoop As Single
Public counter As Long
Sub ElapsedTime
   ' If 0 or negative number of iterations is specified,
   ' print a message and end execution.
   If counter& <= 0 Then
      Print "Number of iterations must be >0"
      End                          ' End execution
   End If
   startLoop! = Timer()
   For doCount& = 1 To counter&
   Next
   Print Timer() - startLoop! "seconds to run" counter& "iterations"
End Sub
Sub DoTimer
   ' DoTimer calls ElapsedTime and reports the result.
   anInt$ = InputBox$("Enter a whole number:")
   counter& = CLng(anInt$)
   startSub! = Timer()
   Call ElapsedTime()
   ' This Print statement will not be executed if the End statement
   ' in sub ElapsedTime was executed.
   Print Timer() - startSub! "seconds for ElapsedTime sub call"
End Sub
Call DoTimer()
' Sample output, for 5000 iterations requested by user:
' .109375 seconds to run 5000 iterations
' .1601563 seconds for ElapsedTime sub call
' Output for -1000 iterations requested by user:
' Number of iterations must be >0
```

In this example, the sub DoTimer is called, which then calls the sub ElapsedTime. When the End statement in ElapsedTime is executed, execution continues at the Print statement following the DoTimer call. Compare this result with the result of the earlier example containing ElapsedTime, which includes the Exit Sub statement.

Do statement

The Do statement executes a block of statements repeatedly while a given condition is true, or until it becomes true. The block of statements executes infinitely often if the condition for termination is never satisfied.

There are three kinds of Do statements. They differ in whether there is a condition or in where the condition appears in the statement. There may be no condition at all, or it may be specified at the beginning, or at the end, using either a While phrase or an Until phrase. The syntax for these three forms is:

- **Do...Loop**

  This form of Do statement includes no condition.

- **Do While** *condition*...**Loop** or **Do Until** *condition*...**Loop**

  In this form, the condition is evaluated before each iteration.

- **Do...Loop While** *condition* or **Do...Loop Until** *condition*

  In this form, the condition is evaluated after each iteration.

The following example, which illustrates the first form of Do statement, appeared in the first example in the section "Exit statement":

```
doCount% = 0
Do
   doCount% = doCount% + 1
   If doCount% >= 1000 Then Exit Do
Loop
```

The Do loop in this example repeats until the condition in the If statement is satisfied. A Do statement like this one, without a While phrase or an Until phrase, must contain an Exit statement or an End statement, or some other statement that transfers control out of the Do statement, such as GoTo. Otherwise the loop runs forever.

In the following example, each Do statement is equivalent to the Do statement in the preceding example:

```
Dim doCount As Integer

' A Do While statement (condition at the beginning)
doCount% = 0
Do While doCount% < 1000
   doCount% = doCount% + 1
Loop

' A Do Until statement (condition at the beginning)
doCount% = 0
Do Until doCount% >= 1000
   doCount% = doCount% + 1
Loop

' A Do...Loop While statement (condition at the end)
doCount% = 0
Do
   doCount% = doCount% + 1
Loop While doCount% < 1000
```

```
' A Do...Loop Until statement (condition at the end)
doCount% = 0
Do
   doCount% = doCount% + 1
Loop Until doCount% > 1000
```

The forms of the Do statement differ with regard to whether the *condition* is tested before or after the first iteration of the loop. The *condition* in a Do While *condition* statement or a Do Until *condition* statement is tested before the first iteration, whereas the *condition* in a Do...Loop While *condition* statement or a Do...Loop Until *condition* statement is not tested until after the first iteration. As a result:

- The body of a Do While...Loop statement or a Do Until...Loop statement may not be executed at all.
- The body of a Do...Loop While statement or a Do...Loop Until statement is executed at least once.

This example shows the difference:

```
Dim doCount As Integer

doCount% = 1
Do While doCount% < 1
   doCount% = doCount% + 1
Loop
Print "Do While...Loop counter reached" doCount%

doCount% = 1
Do
    doCount% = doCount% + 1
Loop While doCount% < 1
Print "Do...Loop While counter reached" doCount%
' Output:
' Do While...Loop counter reached  1
' Do...Loop While counter reached  2
```

The Do statement doesn't establish a separate scope for variables within it. A variable used in a While *condition* clause or an Until *condition* clause is like any other variable in the script. If the variable has not been used previously, then its appearance in *condition* declares it implicitly, and initializes it. For example:

```
' Suppose that the variable named doCount%
' has not appeared in a script prior to its appearance here.
Do While doCount% < 1
   doCount% = doCount% + 1
Loop
Print "Do While...Loop counter reached" doCount%
' Output:
' Do While...Loop counter reached  1
```

LotusScript declares doCount% implicitly and initializes it to 0, so the body of the loop executes once. However, it's risky programming practice to rely on this initialization. You couldn't rely on this behavior without knowing that either doCount% has not appeared earlier during execution, or that the current value of doCount% is 0.

In the next example, a Do statement calculates successive terms of a sequence of numbers that converges to a limit.

```
' This sub computes the quotient of each successive pair of terms
' of the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, ....
' The sequence of quotients 2, 3/2, 5/3, ... is known to converge to
' the golden mean (1 + Sqr(5))/2.
' The sub argument deltaLim! is the tolerance.
' This example illustrates the Do...Loop Until form of the
' Do statement, with a condition that is recomputed on each iteration.
Sub FibiLim (deltaLim As Single)
   Dim r1 As Single, r2 As Single, r3 As Single
   Dim limTrue As Single
   Dim i As Integer
   ' Initialize the Fibonacci numbers and a counter.
   r1! = 1
   r2! = 1
   r3! = 1
   i% = 2
   Do
   NexTerm:
      i% = i% + 1
      r1! = r2!
      r2! = r3!
      ' r3! is the next Fibonacci number.
      r3! = r2! + r1!
      Print i%, "f(" & Str(i%) & "):" r3!, "quotient: " r3! / r2!
      ' On the first iteration, disable the standard exit condition.
      If i% = 3 GoTo NexTerm
      ' Iterate until successive quotients are close.
      ' The sequence is known to converge, so the iteration will end.
   Loop Until Abs(r3! / r2! - r2! / r1!) < deltaLim!
   limTrue! = (1 + Sqr (5)) / 2
   ' When done, show the closeness obtained and the actual limit.
   Print "Tolerance:" deltaLim!
   Print "Difference:" CSng(Abs(r3! / r2! - limTrue!)), _
      "(Actual limit:" limTrue!")"
End Sub
```

```
' Call FibiLim with a tolerance argument.
Call FibiLim(.005)
' Output:
' 3           f(3): 2       quotient:  2
' 4           f(4): 3       quotient:  1.5
' 5           f(5): 5       quotient:  1.66666666666667
' 6           f(6): 8       quotient:  1.6
' 7           f(7): 13      quotient:  1.625
' 8           f(8): 21      quotient:  1.61538461538462
' 9           f(9): 34      quotient:  1.61904761904762
' Tolerance: .005   Difference: 1.013614E-03   (Actual limit:
1.618034)
```

While statement

The While statement executes a block of statements repeatedly while a condition is true. The syntax is:

While *condition*

statements

Wend

LotusScript evaluates the *condition* of a While statement before each repetition of the statement body. As soon as the condition is false, control passes to the statement following Wend.

No statement outside the While statement body should transfer control into it, bypassing the evaluation of *condition*; the results are unpredictable.

The While statement is a historical artifact. It is equivalent to the Do While...Loop statement. You should use the Do While...Loop statement in preference to the While statement.

For statement

The For statement executes a block of statements a specified number of times. The syntax is:

For *countVar* = *first* **To** *last* [**Step** *increment*]

[*statements*]

Next [*countVar* [, *countVar*]...]

The simplest form of a For statement does not use the Step or Next optional items in the syntax, as the following example shows.

```
Dim power2 As Integer
For iV = 1 To 15
   power2 = 2 ^ iV - 1
   Print power2% ;
Next
' Output:
' 1 3 7 15 31 63 127 255 511 1023 2047 4095 8191 16383 32767
```

The first line of the For statement in the previous example is equivalent to the following:

```
For iV = 1 To 15 Step 1
```

That is, if the phrase Step *increment* is omitted from the statement, the default value of *increment* is 1.

Note that the body of the For statement can be empty: there need be no statements at all between For and Next.

Variables in the control expressions: their data type and declaration

If any variables appear in the control expressions *first*, *last*, or *increment*, LotusScript uses their current values. If they were not previously declared or used, LotusScript implicitly declares them as Variants and initializes them to EMPTY. You must be certain that any variables in these expressions have been declared before executing the For statement.

LotusScript initializes the counter variable to the value of *first* when the For statement is entered. If *countVar* was not previously declared or used, LotusScript declares it as a Variant. (Note that if your script includes the Option Declare statement, then *countVar* must be declared before you use it in a For statement.)

For example:

```
' If the variable iV was previously declared or used,
' this For statement declares it as a Variant.
' Its value after the For statement completes execution is the
' last value assigned to it during the For statement execution (16).
For iV = 1 To 15
Next
Print TypeName(iV), iV
iV = "abc"
Print TypeName(iV), iV
' Output:
' INTEGER       16
' STRING       abc
```

In the following example, a compiler error results when you attempt to use 2 ^ 15 as the limiting value for an Integer counter variable in a For statement. This is because the maximum Integer value in LotusScript is (2 ^ 15) - 1.

```
Dim i As Integer
For i% = 1 To 2 ^ 15
Next
' Output:
' Error 6: Overflow
```

When the counter variable is a Variant, LotusScript converts its value to the appropriate data type when it executes the For statement. For example:

```
For iV = 1 To 2 ^ 15
Next
Print TypeName(iV), iV
' Output:
' LONG    32769
```

This example is similar:

```
' The Variant kV has a Double value in every iteration of this loop,
' because the For statement first assigns it the Double value 1.0
' and thereafter adds 1 to the value in each iteration.
For kV = 1.0 To 3
Next
Print TypeName(kV), kV
' Output:
' DOUBLE   4
```

In this example, the value of kV during the second iteration of For is the Double value 2.1:

```
' This loop iterates only twice because the third value of kV is 3.2,
' which is larger than the limiting value, 3.
For kV = 1 To 3 Step 1.1
   Print TypeName(kV), kV
Next
' Output:
' INTEGER   1
' DOUBLE   2.1
```

The LotusScript data type conversion rules apply to the counter variable. For example:

```
' In this instance, the Step value, 1.1, is rounded to the Integer
' value 1 each time it is used to increment k%, because k% is declared
' as an Integer variable.
Dim k As Integer
For k% = 1 To 3 Step 1.1
   Print TypeName(k%), k%
Next
' Output:
' INTEGER       1
' INTEGER       2
' INTEGER       3
```

Nested For statements

The following example illustrates the usefulness of nested For statements. The example computes and prints the binomial coefficients (denoted mathematically b(j; k)) for every integer k from 1 to n, for any positive integer n. The algorithm used is the Pascal triangle method, by which b(j; k) is calculated as the sum b(j - 1; k - 1) + b(j - 1; k).

```
' In this example, three separate For statements are nested
' inside an outer For statement.

Sub CoArray(n As Integer)
   Dim i As Integer, j As Integer, k As Integer
   Dim coHold() As Double, coCalc() As Double
   ' Initialize arrays coHold and coCalc to 0.
   ' Alternate elements within each of these arrays will always be 0.
   ' The coefficients are stored in coCalc by addition from coHold.
   ReDim coHold(2 * n%)
   ReDim coCalc(2 * n% + 1)

   coHold(n%) = 1
   Print "Binomial coefficients for the integers up to:" n%

   ' Each iteration of this outer For statement "For j% ..."
   ' computes a line of coefficients.
   For j% = 0 To n%
      If j% > 0 Then
         ' The statement "For k%..." creates an array of coefficients
         ' in the middle of array coCalc. Alternate elements in this
         ' part of coCalc remain 0, and the ends of coCalc remain 0.
         For k% = n% - j% + 1 To n% + j% - 1
            coCalc(k%) = coHold(k% - 1) + coHold(k% + 1)
         Next k%
      End If
```

```
 ' Set the 0-th and j-th coefficients to 1.
      coCalc(n% - j%) = 1
      coCalc(n% + j%) = 1

      Print
      Print "Coefficients for j = "j%":";
      ' The statement "For k% ..." writes the new coefficients
      ' back into coHold to be used the next time around.
      For k% = n% - j% To n% + j%
         coHold(k%) = coCalc(k%)
      Next k%
      ' This For statement prints the line of coefficients for
      ' this value of j%. Every 2nd element of coCalc is 0.
      ' Don't print 0's.
      For k% = 0 To 2 * n%
         If coCalc(k%) > 0 Then Print coCalc(k%);
      Next k%
   Next j%
End Sub

Call CoArray(5)

' Output:
' Binomial coefficients for the integers up to: 5
' Coefficients for 0 : 1
' Coefficients for 1 : 1  1
' Coefficients for 2 : 1  2  1
' Coefficients for 3 : 1  3  3  1
' Coefficients for 4 : 1  4  6  4  1
' Coefficients for 5 : 1  5  10  10  5  1
```

You can call the sub CoArray with larger argument values to obtain other sets of binomial coefficients.

Some other features of this algorithm are worth mentioning:

- To print the coefficients only for n, rather than for every integer up to n, simply move the final nested For statement (For k% = 0 To 2 * n...) outside of the current outer For statement (For j% = 0 To n...), after the phrase Next j%.
- For small values of n, the algorithm is the easiest way of computing and writing out all of these binomical coefficients by hand in a symmetric triangular array, where the longest, bottom row contains the coefficients for n itself. Each coefficient is the sum of two coefficients already computed: its “northwest” and “northeast” neighbors in the array. For n = 15, say, the left half of the array can be produced by hand addition in a minute or so; the right half is its mirror image.

- If the factorials of 1 through n are known, they can be used to compute the binomial coefficients. If a function to compute the factorial is called FactNum, then a binomial coefficient b(n; k) can be expressed as

  ```
  FactNum(n%) / (FactNum(k%) * FactNum(n% - k%))
  ```

  This is a more conventional way of computing the coefficient. You may want to write a routine using FactNum to compute and print the same set of coefficients generated by the sub CoArray in the example above. FactNum itself can be written as a function using a For statement:

  ```
  Function FactNum(n As Integer) As Double
     FactNum# = 1
     For i% = 1 To n%
        FactNum# = FactNum# * i%
     Next i%
  End Function
  ```

  Each method has its advantages:

  - The formula using FactNum is the definition of the binomial coefficient, so that routine may be easier to read and modify.
  - The implementation by CoArray is fast, and involves no calls to other routines. Also, CoArray can take larger arguments than FactNum, since the largest number CoArray computes is a coefficient, rather than the factorial of n.

The definition of the sub CoArray ends with two Next statements that complete two For statements. You can rewrite the Next statements in this way:

```
   Next k%
Next j%
```

That is, k% and j% are optional in these statements. The following is also equivalent:

```
Next k%, j%
```

When you use this construction, you must order the counter variables correctly: from the inside For statement to the outside.

Common errors in For statements

The following situations represent some straightforward logic errors in writing For statements, and illustrate how LotusScript responds to them.

- Two For statements can be nested, but they cannot overlap partially. For example:

```
For i% = 1 To 3
   For j% = 1 To 2
Next i%
   Next j%
' Output:
' Error 53: Name does not match FOR count variable: I
```

- A For statement cannot overlap with any other block statement. For example:

```
For i% = 1 To 3
   Do
      Print "test"
   Next
Loop
' Output:
' Error 1: Unexpected: NEXT; Expected: LOOP
```

- Within a For statement, its counter variable cannot be used as the counter variable of another For statement. For example:

```
For i% = 1 To 3
   For i% = 1 To 3
   Next
Next
' Output:
' Error 52: FOR count variable already in use: I
```

ForAll statement

This statement executes a block of statements repeatedly, once for each element of an array or a list. The syntax is:

ForAll *refVar* **In** *container*

statements

End ForAll

container names an existing array or list.

After the statements in the body of the ForAll statement are executed for the last element in *container*, execution continues with the next statement following the ForAll statement. However, execution may continue elsewhere if control passes out of the body of the ForAll statement during execution, via a GoTo, GoSub, Exit, or End statement.

On successive iterations of *statements*, the reference variable, *refVar*, refers in turn to each element in *container*. The name *refVar* is declared by its appearance in the ForAll statement. It is not a synonym for the container name itself. Rather, it is an **alias** for each individual element of the container in turn. On each successive iteration, its data type is the data type of the element of the container.

For example:

```
Dim persStats List As String            ' Declare list of type String.
persStats("Name") = "Ian"               ' Assign list elements.
persStats("Age") = "36"
persStats("Home state") = "MD"
ForAll idAttrib In persStats            ' For each item in persStats,
   Print ListTag(idAttrib)": " idAttrib ' print its tag and value.
End ForAll
' Output:
' Name: Ian
' Age: 36
' Home state: MD
```

Here is an example of a ForAll statement where the container is an array instead of a list:

```
Dim numberId(2) As Integer
For i% = 0 To 2
   numberId(i%) = i% + 1
Next
ForAll p2 In numberId
   Print TypeName(p2) p2 * p2  ' Print the type and the square of
                               ' the number in each element.
End ForAll
' Output:
' INTEGER 1
' INTEGER 4
' INTEGER 9
```

If an array or a list has no elements, then a ForAll statement with that array or list for a container variable has no effect. For example:

```
Dim testNone() As Integer
Print "Before ";
ForAll iTest In testNone
   Print iTest, "In ForAll ";
End ForAll
Print "After"
' Output:
' Before After
```

Scope of the reference variable

It is illegal to refer to the reference variable outside the ForAll statement. For example:

```
ForAll p2 In numberId
   Print p2 * p2 ;
End ForAll
Print
Print TypeName(p2)
' Output:
' 1  4  9
' Error 115: Illegal reference to FORALL alias variable: P2
```

It is also illegal to declare a reference variable outside a ForAll statement. For example:

```
Dim p2 As Integer
ForAll p2 In numberId
   Print p2 * p2 ;
End ForAll
' Output:
' Error 164: FORALL alias variable was previously declared: P2
```

You can, however, reuse a reference variable from one ForAll statement as the reference variable in another ForAll statement. The container variable in the second ForAll statement must have the same data type as the container variable in the first ForAll statement. The LotusScript compiler generates an error if the data types are different. (The kind of container — array or list — doesn't matter.)

For example:

```
ForAll p2 In numberId
   Print p2 * p2 ;
End ForAll
Print

Dim numShiftV(3) As Variant
ForAll p2 In numShiftV
   p2 = 1
End ForAll
' Output:
' 1  4  9
' Error 114: FORALL alias variable is not of same data type: P2
```

In the example, p2 was implicitly declared as an Integer variable by the statement:

```
ForAll p2 In numberId
```

Therefore it cannot be redeclared as a Variant variable, as this statement tries to do:

```
ForAll p2 in numShiftV
```

Changing the declared data type of numShiftV to Integer would make the second use of p2 legal.

Modifying container variable elements

The following example illustrates how a ForAll statement references the current value of each element in the container array or list. In the example, statements within the ForAll statement change the current values of the two elements in the container array iHold. The new values are used by subsequent statements in the first iteration of the ForAll statements, and also when the ForAll statements are executed for the next element in iHold.

```
Dim iHold(1) As Integer
iHold(0) = 50
iHold(1) = 100
ForAll iElem In iHold
   ' Print the values of iElem and iHold(1)
   ' upon each entry into ForAll.
   Print
   Print "iElem and iHold(1) IN:" iElem iHold(1)
   ' Add 2 to the current element. The current element is iHold(0) the
   ' first time through ForAll, and iHold(1) the second time through.
   iElem = iElem + 2
   ' Increment the value of iHold(1) by 5 (both trips through).
   iHold(1) = iHold(1) + 5
   ' Print the current values of iElem and iHold(1)
   ' upon each exit from ForAll.
   Print "iElem and iHold(1) OUT:" iElem iHold(1)
End ForAll
' Output:

' iElem and iHold(1) IN: 50  100
' iElem and iHold(1) OUT: 52  105

' iElem and iHold(1) IN: 105  105
' iElem and iHold(1) OUT: 112  112
```

To compare how a With statement can perform a similar task, see the description of With in Chapter 5, “User-Defined Data Types and Classes.”

In the following example, the value of an element of the container array cHold is a reference to an object of the class refClass. Initially the two elements of cHold contain different object references. On the first iteration of the ForAll statement, the value of the first element is reset to the value of the second; thereafter, the elements refer to the same object.

```
Option Base 1
Class refClass
   Public cVar As Integer
End Class
Dim cHold(2) As refClass
Set cHold(1) = New refClass
Set cHold(2) = New refClass
' The output from the following Print statement
' shows that cHold(1) and cHold(2) hold different objects references.
If cHold(1) Is cHold(2) _
   Then Print "Same object" Else Print "Different objects"
cHold(1).cVar% = 100
cHold(2).cVar% = 200
ForAll cElem In cHold
   Print
   Print cElem.cVar%
   Set cHold(1) = cHold(2)
   ' Now cHold(1) holds the same reference as cHold(2), so
   ' cElem refers to that object in the following statements
   ' (on both trips through ForAll).
   Print cElem.cVar%
   If cHold(1) Is cHold(2) _
      Then Print "Same object" Else Print "Different objects"
End ForAll
' Output:
' Different objects
'
' 100
' 200
' Same object
'
' 200
' 200
' Same object
```

The two examples above change the contents of the container array for the ForAll statement, but not the structure. Since the container is the control structure for a ForAll statement, it's a very questionable programming tactic to change the structure. Although you can use the Erase statement on the container or its elements; or use the ReDim statement on an array, this is not recommended. The results are unpredictable.

Similarly, it is possible to transfer control from outside a ForAll statement to a labeled statement inside. But this is also not recommended, since by doing so you bypass the built-in initialization of the ForAll reference variable that occurs when the ForAll statement begins execution for a particular element.

Element access order

As shown in the first example in this section, a ForAll statement for a list container accesses the list elements in the same order as they are maintained in the list. A ForAll statement for an array accesses the array elements in the order in which LotusScript stores them. For a one-dimensional array arrA, this is arrA(0), arrA(1), arrA(2), ... (if 0 is the lowest subscript for arrA). LotusScript stores an array with more dimensions in **first-fastest** order (the first subscript in the array subscript list varies fastest). A ForAll statement accesses the array elements in the same order. For example:

```
Option Base 1g5

Dim eyeJay(2,3) As String
' Access the elements of eyeJay in "last fastest" order
' for assignment and printing.
For i% = 1 To 2
   For j% = 1 To 3
      ' In eyeJay(i,j), store the string "(i,j)".
      eyeJay(i%, j%) = "(" & Str(i%) & "," & Str(j%) & ")"
      ' Print the element value.
      Print eyeJay(i%, j%),
Next j%, i%
Print
' Now print the elements of eyeJay one at a time in the same order
' as the ForAll statement accesses them.
' This order is first fastest, the storage order for any array.
Print
ForAll elem In eyeJay
   Print elem,
End ForAll
' Output:
' ( 1, 1)  ( 1, 2)   ( 1, 3)   ( 2, 1)   ( 2, 2)   ( 2, 3)
' ( 1, 1)  ( 2, 1)   ( 1, 2)   ( 2, 2)   ( 1, 3)   ( 2, 3)
```

Chapter 8
Error Processing

Two kinds of errors can occur in a LotusScript application: compile-time errors and run-time errors.

Compile-time errors are errors that are found and reported when the compiler attempts to compile a script. Common compile-time errors are errors in the syntax of a statement, or errors in specifying the meaning of names: for example, misstating the bounds of an array variable.

The compiler reports an error, together with an error message and a link to online Help, which explains how to correct the error. You need to correct any such error by revising the script source statements that generated the error before the script can compile and run.

Run-time errors occur when LotusScript attempts to execute the script. A run-time error represents a condition that exists when the script is run, but could not be predicted at compile time. Examples of run-time errors are attempting to open a file that doesn't exist, or attempting to divide a number by a variable whose value is zero (0).

Run-time errors prevent a script from running to normal completion. When a run-time error occurs, script execution ends unless your script includes statements to handle the error.

LotusScript anticipates many run-time errors, and identifies each one with a number, a name, and a standard message that describes it. You can also create your own errors and associate a number and a message with each one.

This chapter covers the following topics in run-time error processing:

- Managing run-time errors
- Using the On Error and Resume statements
- Using error-handling routines outside procedures
- Using the informational functions

Managing Run-Time Errors

This section summarizes statements and functions you use in your application to manage run-time errors. It also summarizes how LotusScript handles errors, depending on how you design your application.

The On Error and Resume statements

You include On Error and Resume statements in a script to explicitly manage the flow of control when an error occurs.

- The On Error statement specifies how to handle an error.
- The Resume statement specifies where to resume script execution after handling an error.

How the On Error and Resume statements work together is described in two later sections of this chapter. For a summary of these statements, see "How errors are handled." For details and examples, see "Using the On Error and Resume statements."

Informational functions: Err, Erl, Error, and Error$

The functions Err, Erl, Error, and Error$ describe the current error, if there is one. LotusScript assigns a value to each of these functions when an error occurs.

- The Err function returns the LotusScript error number for the current error, or the number you specify with the Err statement.
- The Erl function returns the number of the line in which the current error occurred.
- The Error and Error$ functions return the error message for the current error, or the message for the error number you specify with the Err statement.

These functions are illustrated in passing throughout this chapter. For examples that specifically illustrate these functions, see "Using the informational functions" at the end of this chapter.

Managing the error number and message: the Err and Error statements

The Err statement sets the error number. The Error statement generates an error, and optionally specifies an error message for it.

The Err statement

The Err statement corrresponds to the Err function, which returns the current error number.

The Err statement has the form:

Err = *errNumber*

The error number can be set either automatically by LotusScript, when an error occurs, or explicitly by this statement in a script. Whenever the error number is set, LotusScript automatically sets the value of the Error function to the error message associated with that error number. If the error number is set to 0, LotusScript sets the value of the Error function to its initial value, the empty string ("").

Note that the Err statement does not create an error as the Error statement does (see below). It only resets the error number (and therefore the value also of the Error function). So the error number Err may be nonzero while there is no current error.

The Error statement

The Error statement creates an error, and optionally specifies an error message associated with that error. The statement has the form:

Error = *errNumber* **[** , *msgExpr* **]**

If you do not include the optional *msgExpr* string in the statement, it simply creates an error when the script runs. If *errNumber* is the number of an error that is already defined, then the effect of this statement is the same as if that error occurred when the script executed. For example, LotusScript defines a division-by-zero error with the error number 11. So the following statement has the same effect as an actual error occurring when LotusScript executes a statement that attempts to divide by zero:

Error = 11

If you include *msgExpr* in the Error statement, you specify the error message to be reported when the error occurs and no error handling for the error is in effect.

How errors are handled

At any time during execution, there is either a **current error**, or no error at all. The current error is a run-time error that has occurred, but has not yet been handled. For the error to be handled in the current procedure, the procedure must include an On Error statement that refers to the error, and that has already been executed.

Using On Error GoTo *label*

When the most recently executed On Error statement for the current error has the form On Error GoTo *label*, LotusScript continues execution at the labeled statement. The statement begins an **error-handling routine** for the error. The error-handling routine may consist of any statements, beginning with the statement executed at the label and continuing through the next Resume, Exit Sub, Exit Function, Exit Property, or End statement encountered at run time. The error is considered handled when one of the latter statements is executed.

If the statement that ends the error-handling routine is a Resume statement, then the values of Err, Erl, and Error are reset to their initial values: 0, 0, and the empty string (""), respectively. If the statement is Exit Sub, Exit Function, or Exit Property, then LotusScript does not reset the values of the Err, Erl, and Error functions.

Using On Error Resume Next

When the most recently executed On Error statement for the current error has the form On Error Resume Next, LotusScript resumes execution with the statement following the statement where the error occurred. When execution resumes in this way, the error is considered handled. LotusScript does not reset the values of the Err, Erl, and Error functions that were set when the error occurred.

Handling an error outside a procedure

If the current procedure contains no On Error Resume Next statement or On Error Goto *label* statement that refers to the error, or if such a statement was not executed during the current call to the procedure, LotusScript determines whether one of these statements was executed in the procedure that called this procedure, if any. If so, the error is handled as specified by that statement, as described above. If not, the search for an applicable On Error statement continues in the procedure that called that procedure, and so on. If no associated On Error statement was executed in any calling procedure, then execution ends and the associated error message is displayed.

An On Error statement of the special form On Error GoTo 0 does not handle any error that it refers to. It says explicitly that any error it refers to is *not* handled in the current procedure. When such an error occurs, LotusScript searches upward through the chain of calling procedures for an On Error statement that specifies how to handle the error.

Resetting the error number

The value of the function Err persists across scripts. Completing execution of a script does not automatically reset this function's value to 0. The value of Err is reset to 0 only by an Err statement or a Resume statement.

Errors within error-handling routines

If an error occurs during execution of an error-handling routine, that error becomes the current error. Execution ends and the associated error message is displayed.

Using the On Error and Resume Statements

To handle a run-time error, you can write an error-handling routine, the set of statements that are executed when the error occurs. Ordinarily, an error-handling routine performs some special action related to the error, and then execution resumes and continues normally.

You can specify that an error-handling routine handles only a particular error in the script, or any run-time error at all in the script. You can write several error-handling routines, each to handle a particular error, and also a general error-handling routine to handle all the remaining possible errors.

The following extended examples show how a run-time error can arise in a script, and how you can modify a script to either avoid or handle the error. The straightforward error processing illustrated here uses the On Error and Resume statements, which you typically use to process errors.

Suppose that your script includes a sub named GetLine to retrieve some values from the first line of a file whose name the user specifies. For example:

```
Sub GetLine
   Dim number1 As Integer, number2 As Integer, number3 As Integer
   Dim fileName As String
   ' Prompt the user to enter a file name, and assign the result.
    fileName$ = InputBox$("Enter a file name: ")
   Open fileName$ For Input As #1      ' This is line 6.
   Input #1, number1%, number2%, number3%
   Print number1%, number2%, number3%    ' Print the input values.
   Close #1
End Sub
```

When the sub GetLine runs, an error occurs at the Open statement if the user enters the name of a nonexistent file when prompted by the InputBox$ function. Because the script does not contain statements to handle the error, LotusScript ends execution of the script and prints an error message:

```
Call GetLine()
' Output:
' Fail: RunTime Error 101 Unable to open file at Line 6
```

In the following example, the script just shown is modified to include an On Error statement to handle a file-open error when it occurs. If the Open statement fails, LotusScript prints some identifying information about the error, and requests a new file name from the user, rather than ending script execution and printing an error message.

```
Sub GetLine
   Dim number1 As Integer, number2 As Integer, number3 As Integer
   Dim fileName As String
   ' Designate an error-handling routine to handle an error.
   On Error GoTo NoExist
GetName:
   fileName$ = InputBox$("Enter a file name: ")
   Open fileName$ For Input As #1    ' This is line 8.
   Input #1, number1%, number2%, number3%
   Print number1%, number2%, number3%
   Close #1
   ' Done. Exit from the sub GetLine. (Don't continue on to the
   ' error-handling routine at the label NoExist.)
   Exit Sub
```

```
NoExist:
   ' Come here when any error occurs.
   ' Print the values of built-in functions that give information
   ' about the error: an error message, the error number,
   ' and the line number in the script where the error occurred.
   Print Error(), Err(), Erl()
   ' Resume execution at the label GetName.
   Resume GetName
End Sub
Call GetLine()
' The user twice enters a file name that doesn't exist, and then a
' valid file name. The values read in from the file are 11, 22, and 0.
' Output:
' Unable to open file         101           8
' Unable to open file         101           8
' 11            22            0
```

Error-number constants

The On Error statement in the preceding example designates an error-handling routine to be invoked when any run-time error occurs. However, the error-handling routine was intended only to handle a file-open failure. Therefore, the On Error statement specifies that particular error. The error-handling routine is invoked when that error occurs, but script execution ends when any other error occurs.

A file-open failure is one of the common run-time errors for which LotusScript defines an error message, an error number, and a constant whose value is that number. The constants and their values are defined by Const statements in the file LSERR.LSS. When you include this file in your script, using a %Include directive, you can use the constants to designate errors in your script. The constants themselves are mnemonic names for the errors. For example, the constant specifying the file-open failure is ErrOpenFailed. The file LSERR.LSS itself contains all the run-time errors defined by LotusScript. Since the file specifies a number for each constant, you can use the error number in an On Error statement. This is not advisable, however, because the number is less meaningful than the name of the constant.

You can modify the previous example to include LSERR.LSS at the beginning of the script, and replace the On Error statement by this line:

```
On Error ErrOpenFailed GoTo NoExist
```

When the modified script runs, the error-handling routine at the label NoExist is invoked when the error ErrOpenFailed occurs. Script execution ends when any other error occurs. For example, script execution ends if the first three values in the file do not have the data type Integer, because the Input # statement generates an error.

Multiple On Error statements

Handling individual errors

An On Error statement can refer to only one error-handling routine. To specify that you want different errors handled differently, you need to include two or more On Error statements in your script.

For example, you might modify the example in "Using the On Error and Resume statements," earlier in the chapter, to include a Print statement that can generate a division-by-zero error. To handle a division-by-zero error, you could include an additional On Error statement that specifies this error and designates an error-handling routine that responds appropriately to the error. The routine begins at the DivZero label. It includes an InputBox$ function call that prompts the user to type a replacement value for the 0 (zero) that was read from the opened file. The additional On Error statement is

```
On Error ErrDivisionByZero GoTo DivZero
```

The error-handling routine looks like this:

```
DivZero:
   number3% = InputBox$("Number3 is 0. Enter a new value: ")
   ' Resume execution with the statement that caused
   ' the error ErrDivisionByZero.
   Resume
```

When the above statements are added, the script manages file-open failure errors and division-by-zero errors. However, any other error terminates script execution. To ensure that all other errors are handled without terminating script execution, include an On Error statement that doesn't specify a particular error.

With all of these changes, the resulting script looks like this:

```
%Include "LSERR.LSS"
Sub GetLine
   Dim number1 As Integer, number2 As Integer, number3 As Integer
   Dim fileName As String
   ' The error-handling routine at label Leave is for
   ' all errors except the two individual errors
   ' specified in the second and third On Error statements.
   ' Each has a specific error-handling routine designated.
   On Error GoTo Leave
   On Error ErrOpenFailed GoTo NoExist
   On Error ErrDivisionByZero GoTo DivZero
GetName:
   fileName$ = InputBox$("Enter a file name: ")
   Open fileName$ For Input As #1
   Input #1, number1%, number2%, number3%
   Print number1%, number2%, number3%
```

```
' The next statement causes a division-by-zero error if number3 is 0.
   Print (number1% + number2%) / number3%
   Close #1
   Exit Sub
NoExist:
   Print Error(), Err(), Erl()
   Resume GetName
DivZero:
   number3% = InputBox("Number3 is 0. Enter a new value: ")
   Resume
Leave:
   ' The following message is general, because different errors
   ' may have occurred.
   MessageBox("Cannot complete operation.")
   Exit Sub
End Sub
```

The following example of a call to GetLine shows how the modified sub works:

```
Call GetLine()
' The user enters a valid file name, and the values read in
' from the file are 11, 22, and 0.
' Output:
' 11           22            0
' The value 0 causes a division-by-zero error.
' The user then enters the value 2 into the input box specified
' in the error-handling routine beginning at DivZero.
' Execution resumes at the Print statement that generated the error.
' Output:
' 16.5
```

For all errors other than file-open failure errors and division-by-zero errors, the error-handling routine at Leave displays a message in a message box, and returns from the sub GetLine.

However, suppose that the user enters the value 99999, instead of the value 2, into the input box in the error-handling routine at DivZero. The result is an overflow error, because 99999 is larger than the maximum legal Integer value for the variable number3%. This error will not be handled, because it occurs within the error-handling routine at DivZero. LotusScript ends execution whenever an error occurs within an error-handling routine.

Ordering of On Error statements

The order of On Error statements is important. Only one error-handling routine (or none) is in effect at any given time for any particular error. The one in effect is the routine specified in the most recently executed On Error statement that applies to that error. Changing the order of the On Error statements can change the processing at run time. For example, suppose the order of the three On Error statements at the beginning of the preceding example is changed to this:

```
' Two routines are designated to handle individual errors.
On Error ErrOpenFailed GoTo NoExist
On Error ErrDivisionByZero GoTo DivZero
' The Leave routine is intended to handle all other errors.
On Error GoTo Leave
```

This sequence of three statements is a programming mistake. After these three statements execute, all errors are handled by the error-handling routine beginning at the label Leave, because the statement On Error GoTo Leave refers to all errors. The routine named Leave overrides the routines established for ErrOpenFailed and for ErrDivisionByZero that were specified in the preceding two On Error statements.

On Error Resume Next

Instead of specifying an error-handling routine that executes when an error occurs, you can specify that program execution simply continues with the next statement after the statement that generates the error. This is done by including the statement On Error Resume Next, as in the following example:

```
Sub TestHand
   Dim num As Single
   On Error Resume Next
   num! = 1
   ' The next statement generates an error.
   Print num! / 0
   Print "Continuing after division-by-zero error."
End Sub
Call TestHand()
' Output:
' Continuing after division-by-zero error.
```

Error-Handling Routines Outside Procedures

When an On Error statement specifies the label where the error-handling routine begins, that labeled statement must be in the same procedure as the On Error statement. This is because a GoTo statement cannot transfer control to a labeled statement outside the procedure where it occurs. The compiler verifies that the labeled statement is present in the same procedure, and generates a compile-time error if it is not.

However, LotusScript need not handle an error in the procedure where it occurs. The error can be handled in the procedure that called the current procedure. If the current procedure doesn't handle the error, LotusScript returns control to the calling procedure and seeks an error-handling routine there for the error. If the caller doesn't handle the error, LotusScript looks at the caller's caller, and so on. If no applicable error-handling routine is found by this process, execution ends, and the error message for the error is generated. For example:

```
' The sub TestHand generates a division-by-zero error.
' Since TestHand doesn't specify how to handle the error, control
' returns to the calling procedure SuperHand when the error occurs.
' SuperHand contains an error-handling routine for division by zero.
' Control passes to that routine, which prints a message and exits
' from SuperHand.
Sub TestHand
   Dim num As Single
   num! = 1
   Print num! / 0
End Sub
Sub SuperHand
   On Error GoTo DivZero
   Call TestHand()
   Exit Sub
DivZero:
   Print "Continuing after calling sub TestHand."
   Exit Sub
End Sub
Call SuperHand()
' Output:
' Continuing after calling sub TestHand.
```

You can use a special form of the On Error statement to state explicitly that a specified error not be handled in the current procedure. The statement has the form:

On Error *errNumber* **GoTo 0**

This says that the error numbered *errNumber* is not handled in the current procedure. For example, the result of the preceding example is unchanged if the sub TestHand is modified as follows:

```
Sub TestHand
   Dim num As Single
   On Error ErrDivisionByZero GoTo 0
   num! = 1
   Print num! / 0
End Sub
```

You can also use a statement in the following form to specify that no error be handled in the current procedure. This statement makes it explicit that the procedure handles no errors, so your error-handling logic is clearer.

On Error GoTo 0

Like any On Error statement, the effect of this statement can be overridden, for any particular error, by a subsequent On Error statement that designates different handling for that error. For example:

```
' This pair of On Error statements specifies that
' division-by-zero errors are handled by an error-handling routine
' at the label DivZero; and no other errors are
' handled in the current procedure (an error-handling routine
' for other errors is sought in the procedure's caller).
On Error GoTo 0
On Error ErrDivisionByZero GoTo DivZero
```

Resuming execution in a calling procedure

When an error is handled that occurred in a lower-level procedure, On Error Resume Next has a special meaning. LotusScript considers the procedure call to be the statement that caused the error; so "Next" refers to the next statement in the calling procedure. For example:

```
Sub TestHand
   Dim num As Single
   num! = 1
   Print num! / 0
End Sub
Sub SuperHand
   On Error Resume Next
   Call TestHand()
   ' When control returns to SuperHand upon an error in TestHand,
   ' execution continues at this Print statement.
   Print "Continuing after calling sub TestHand."
   Exit Sub
End Sub
Call SuperHand()
' Output:
' Continuing after calling sub TestHand.
```

Similarly, when the statement Resume Next appears within an error-handling routine for an error that occurred in a lower-level procedure, "Next" refers to the next statement in the calling procedure.

The statement Resume 0, or simply Resume, in an error-handling routine means to call again the procedure that produced the error, as the following example shows:

```
' The sub SuperHand calls the sub TestHand with an argument of 0,
' which produces an error. The error is handled by an error-handling
' routine in the caller, the sub SuperHand.
' Handling the error includes resetting the call argument to 1,
' and then calling TestHand with this argument. On the second call
' no error occurs.

Sub TestHand(num As Integer)
   Dim num2 As Single
   If num <> 0 GoTo ProcPos
   Print "Call argument to sub TestHand is 0; will generate error."
   ' There's no error-handling routine in sub TestHand for
   ' division-by-zero, so control returns to the calling sub
   ' SuperHand when the next statement is executed.
   num2! = num% / 0
   ' This Print statement is not executed at all.
   Print "Continue here after division-by-zero error?"
   Exit Sub
   ' Come here if call argument is nonzero.
ProcPos:
   Print "Call argument to sub TestHand is nonzero (no error)."
   Exit Sub
End Sub
Sub SuperHand
   Dim numIn As Integer
   ' A division-by-zero error not handled in sub TestHand
   ' is handled by the error-handling routine at DivZero.
   On Error GoTo DivZero
   Call TestHand(numIn%)
   Exit Sub
DivZero:
   Print "Handling division-by-zero error."
   numIn% = 1
   ' Re-execute the statement that caused the error being handled.
   ' This will be the statement Call TestHand(numIn%) above.
   ' The call argument is now 1.
   Resume 0
End Sub
Call SuperHand()
' Output:
' Call argument to sub TestHand is 0; will generate error.
' Handling division-by-zero error.
' Call argument to sub TestHand is nonzero (no error).
```

Using the Informational Functions

The examples in this section illustrate how LotusScript manages the error number and its associated error message and line number.

```
' When the sub DemoErr is called, the values of Error(), Err(), and
' Erl() are assumed to be the empty string (""), 0, and 0
' respectively.
' The occurrence of an error resets them. Completing the associated
' error-handling routine resets them to the initial values.
Sub DemoErr
   ' Show values on entry to sub DemoErr.
   Print "Error: " Error(), "   Err:" Err(), "   Erl:" Erl()
   ' Designate an error-handling routine; then create an error.
   On Error GoTo ShowErr
   Error 11                     ' This is line 10.
   ' Come here after Resume.
   Print "Error: " Error(), "   Err:" Err(), "   Erl:" Erl()
   Exit Sub
ShowErr:
   ' Display the values on entry to the error-handling routine.
   Print "Error: " Error(), "   Err:" Err(), "   Erl:" Erl()
   Resume Next
End Sub
Call DemoErr()
' Output:
' Error:          Err: 0        Erl: 0
' Error: Division by zero       Err: 11       Erl: 10
' Error:          Err: 0        Erl: 0
```

The next example illustrates the flow of control and the change in the values of the control variables Error, Err, and Erl during error processing. Though it will run and behave exactly as shown here, this is an artificial script. It is constructed solely to demonstrate these error-processing features.

```
' This example omits the Exit Sub statement of the preceding example.
' As a result, execution continues on to the error-handling routine.
Sub ShowErr
   On Error GoTo CheckErr
   Error 150                    ' This is line 5.
   Print "Error was handled... Error, Err, Erl are now:"
   Print Error(), Err(), Erl()    ' This is line 7.
   ' Exit Sub statement was dropped here.
CheckErr:
   Print Error(), Err(), Erl()
   Resume Next                  ' This is line 11.
End Sub
```

```
Call ShowErr()
Print "Back from call of ShowErr"
```

After error 150 occurs at line 5, the error-handling routine at CheckErr prints this line:

```
Cannot find module %s       150           5
```

After the Resume statement, the Print statements in lines 6 and 7 prints these two lines:

```
Error was handled... Error, Err, Erl are now:
             0             0
```

Execution continues on normally to the Print statement at CheckErr, which prints the following line:

```
             0             0
```

Execution then continues normally to the Resume Next statement on line 11. Since there is no current error, there is no "Next" statement, so the Resume statement itself is invalid and generates an error, which becomes the current error; and the error-handling routine at CheckErr is invoked again. It prints the following line:

```
RESUME without error        20            11
```

The error-handling routine ends with the statement Resume Next. The "next" statement is End Sub. So the sub exits normally, and the Print statement after the sub call prints the following line:

```
Back from call of ShowErr
```

That completes execution of this example.

In the next example, an Err statement is placed at the beginning of the error-handling routine shown in the preceding example. The result is to invalidate the value of Erl: it no longer describes the error specified by Err.

```
Sub ShowErr
   On Error GoTo CheckErr
   Error 150                      ' This is line 3.
   Print "Error was handled... Error, Err, Erl are now:"
   Print Error(), Err(), Erl()    ' This is line 5.
CheckErr:
   ' Reset the error number, without creating an error.
   Err 151
   Print Error(), Err(), Erl()
   Resume Next                     ' This is line 10.
End Sub
Call ShowErr()
Print "Back from call of ShowErr"
```

After error 150 occurs at line 3, the error-handling routine starting at CheckErr executes. It first sets the error number (the value of Err) to 151. This resets the Error function also (but not the Erl function). So the Print statement prints the following line:

```
Cannot find external name   151           3
```

After the Resume statement, the Print statements on lines 4 and 5 print these two lines:

```
Error was handled... Error, Err, Erl are now:
             0            0
```

Execution continues normally to the statements starting at CheckErr. The Err statement there resets the error number, and the Print statement therefore prints the following line. (Note that there is no current error, and therefore the value of Erl is still 0.)

```
Cannot find external name   151           0
```

The next statement executed, Resume Next, is invalid because there is no current error. So it generates an error, and the error-handling routine beginning at CheckErr is invoked again. It first sets Err to 151, and then prints the following line. (The values of Error and Err represent the latest assignment to Err; but Erl is still 10 because the current error occurred at line 10.)

```
Cannot find external name   151           10
```

The error-handling routine ends with the statement Resume Next. The “Next” statement is End Sub. So the sub exits normally, and the Print statement after the sub call prints the following line:

```
Back from call of ShowErr
```

That completes the execution of this example.

Chapter 9
Reaching Out

This guide has concentrated on providing you with a conceptual overview of LotusScript as a complete and self-contained programming language. Now it is time to turn to the role that LotusScript can play in the larger world of Lotus products, your operating environment, other programs, and interactive user applications.

This chapter covers the following topics:

- The extensions to the LotusScript language supplied by Lotus products
- Using input boxes and message boxes to interact with users
- Reading and writing files
- Using environment variables
- Starting, activating, sending keystrokes to, and yielding control to other programs
- Using Dynamic Data Exchange (DDE) to exchange data and send instructions to DDE server applications
- OLE automation
- Incorporating functionality provided by libraries of external C functions

Working with Lotus products

The Lotus product in which you are working provides the environment in which you create, debug, and run LotusScript modules, so consult your product documentation before you begin using LotusScript.

Each Lotus product that works with LotusScript also supplies its own application programming interface (API), which lets you use product functionality and create and manipulate product objects from within LotusScript. A product API is effectively an extension to the LotusScript language that is available when you are running that product.

Product classes and objects

Each Lotus product with which you use LotusScript provides a number of predefined classes. In many respects, product objects (instances of product classes) are like user-defined objects (instances of user-defined classes). For information about user-defined classes, see Chapter 5, “User-Defined Data Types and Classes.”

But while user-defined objects exist within the scope of LotusScript modules, many product objects have their own existence apart from the scripts in which you manipulate them. You may, for example, use the product interface rather than a script to create, name, and put text on a command button. You can then attach a script to the command button "click" event. When the user clicks the command button, the appearance of the command button changes, and the "click" event script executes.

You can create and assign variable references to product objects, get and set product object properties, use product object methods, attach scripts to product object events, and delete product objects. For detailed information about how to perform all of these operations with specific Lotus product classes, consult the appropriate Lotus product documentation.

Creating objects

In some cases, the product automatically creates objects (cells in a spreadsheet for example). In other cases you use the product user interface to create objects, and in still other cases, you can create objects in a script.

To create an object in a script, you must supply whatever arguments the product requires to create an instance of the particular class, and you must assign an object reference to a variable. In many cases, the syntax is as follows:

Dim *objRef* **As** *prodClass*

Set *objRef* = **New** [*prodClass*] [(*argList*)]

The Dim statement declares an object reference variable. The Set...New statement creates a product object and assigns the variable a reference to that object. You can also combine these operations in a single statement:

Dim *objRef* **As New** *prodClass* [(*argList*])]

In some cases, you use a method to create the object. In Lotus Notes® Release 4, for example, you use the NotesDatabase Create method to create a new .NSF file.

In other cases, you use a container method to create objects in scripts. A container method applies to the object that contains the object you are creating. Freelance Graphics® for Windows, for example, provides container methods for creating objects.

Referring to objects

The process of creating an object in a script involves assigning a reference to the object (see above). To refer in a script to an object that already exists, in some cases you can use the name that the product or user gave to the object. You generally can (and in some cases you must) assign your own object reference.

One way to assign your own object reference to a variable is to declare an object variable:

Dim *objRef* **As** *prodClass*

and bind it to the product object:

Set *objRef* = **Bind** [*prodClass*] [(*objName*])]

In some cases, the product supplies a function or method that you can use to set an object reference.

The following Initialize sub works with three Notes objects: a database, a view, and a document. The sub uses a Dim...New statement to create a new NotesDatabase object to work with ORGSTRUC.NSF on the HR_ADMIN server, and it uses methods in Set statements to set variable references to a view and a document. GetView is a NotesDatabase class method, and GetFirstDocument is a NotesView class method.

```
Sub Initialize
   Dim db As New NotesDatabase("HR_ADMIN", "ORGSTRUC.NSF")
   Dim view As NotesView, doc As NotesDocument
   Set view = db.GetView("Main View")
   Set doc = view.GetFirstDocument
End Sub
```

Bracket notation

In some cases, you can use names in brackets rather than object reference variables to identify Lotus product objects. For example, the product might allow you to use:

```
[A1].Value = 247000
```

instead of:

```
Dim myCell As Cell
Set myCell = Bind Cell("A1")
myCell.Value = 247000
```

For more information, see “Bracket Notation” in Chapter 7 of the *LotusScript Language Reference*, and consult your product documentation.

Properties, methods, and events

Each product class defines a set of properties, methods, and events. As with user-defined classes, you use dot notation to specify properties and methods. For more information about dot notation, see “Dot Notation” in Chapter 7 of the *LotusScript Language Reference*.

Properties are object attributes. Like variables, properties have values. In many cases, you can get and set a property’s value just as you get a value from a variable and assign a value to the variable. Some properties you can only get, and some you can only set.

A Form object, for example, is an instance of the Lotus Forms Form class. It has a number of properties, including FullName, CreationDate, and UseNotesSendTo.

The value of the FullName property is a string specifying the path and file name of the file in which the form is saved. In a script, you can get and set the value of FullName.

The CreationDate property is a date/time value that identifies when the form was created. You can only get the value of the CreationDate property. The following statement, gets the creation date and uses the LotusScript CDat function to store it in a Variant (startDateV) as a date/time value.

```
startDateV = CDat(myForm.CreationDate)
```

You can only set the value of the UseNotesSendTo property. UseNotesSendTo is a flag that you can set to TRUE or FALSE, specifying whether a form embedded in a Notes mail document can be routed through Notes.

The Send method saves a form in a temporary file and sends the form to a mailing address. For example:

```
myForm.Send("Elizabeth Blaney")
```

When the Send method is executed, it causes the SaveForm event to occur.

Events are object-related actions to which you can attach scripts to perform activities in an application. When the event occurs, the script attached to the event executes. For example, you might want to set the value of the FullName property in the SaveForm event script:

```
myForm.FullName = "c:\designer\work\orders.1fm"
```

Lotus products normally handle the process of attaching scripts you write to the events you specify. You can also use LotusScript On Event statements to attach subs to object events.

Deleting objects

In some cases, the objects you use the Lotus product user interface to create are saved from one session to the next, and the objects you create in scripts are temporary. Like any variables, the object reference variables that you explicitly declare and bind to product objects have a scope. When all object references (there may be more than one) to an object created in a script are out of scope, the object itself may be deleted.

In some cases, you can use the Delete statement to delete a product object. In other cases, the Delete statement deletes the object reference variable, but not the unerlying object itself. Some products supply methods to remove actual objects. In Notes, for example, you use the NotesDatabase class Remove method to delete an .NSF file.

Collection classes

Some Lotus products provide collection classes, also known as container classes. A collection object (an instance of a collection class) contains a collection of objects.

In Freelance Graphics, for example, an Application object contains an instance of the Documents collection class. You use the Application class Documents property to return an instance of the Documents collection class. Each element in the collection is a document, an instance of the Document class.

Each Document object contains an instance of the Pages collection class. You use the Document class Pages property to return an instance of the Pages collection class. Each element in the collection is a page, an instance of the Page class.

Each Page object contains an instance of the ObjectCollection class. You use the Page Objects property to return an instance of the ObjectCollection class. Each element in the collection is an instance of the DrawObject class. The ObjectCollection object can include text boxes, charts, tables, and other objects belonging to classes derived from the DrawObject class. Text boxes, for example, are instances of the TextBox class, which is derived from the DrawObject class.

For a discussion of deriving classes (also known as class inheritance), see "Defining derived classes" in Chapter 5.

You can use ForAll loops or indexing to access individual members of a collection class. The following script uses three nested ForAll loops to iterate through the collections. Within individual TextBlock objects, the script uses indexing to set list entries levels 2 through 5 in each TextBox object to italic.

```
Dim level As Integer
ForAll doc In CurrentApplication.Documents
   ForAll page In Document.Pages
      ForAll obj In Page.Objects
         ' If the object is a TextBlock, set the font
         ' to Garamond, and set list entries at levels
         ' 2 through 5 to Italic.
         If obj.IsText Then ' IsText is a DrawObject property.
            obj.Font.FontName = "Garamond"
            For level% = 2 to 5
               obj.TextProperties(level%).Font.Italic = TRUE
            Next level%
         End If
      End ForAll
   End ForAll
End ForAll
```

Determining which product file is being used

On the Windows platform, and to some degree on other platforms, you can use command-line arguments (in the Windows Program Manager for example) to start programs and open product files.

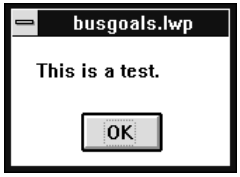
The Command function returns the command-line arguments used to start the Lotus product from which LotusScript was started. If you are starting your Lotus product in such a manner, you may want to use the Command function to get the name of the product file. For example, you may use the file name to identify which product file is currently running, or to provide input for messages to the user.

Suppose, for example, that the command line for launching a Word Pro™ application is

c:\wordpro\wordpro.exe c:\wordpro\docs\busgoals.lwp

The Command function returns "busgoals.lwp". You could make this string the title that appears in any message boxes the script displays.

```
Dim message As String, messageTitle As String
messageTitle$ = Command$
...
...
' Use messageTitle$ as the title of a message box.
message = "This is a test."
MessageBox message$, messageTitle$
```



Interacting with the User

Lotus products lend themselves to building interactive applications, applications that incorporate user input and prompt the user to perform particular tasks. Individual Lotus products provide their own user interface for interacting with scripts. The LotusScript language itself, however, supplies a couple of fundamental tools that you can use with any Lotus product to interact with the user.

You can use the InputBox function to get user input. The InputBox function displays a dialog box with the prompt you specify, a text box, and OK and Cancel buttons. You can also specify a title, a default value, and a position on the screen for the input box.

The user enters characters in the text box and clicks OK. InputBox returns the string the user entered. You can use the data type conversion functions (DateValue, CCur, CDat, CDbl, CInt, CLng, CSng, CVar) to convert the input to a numeric, date/time, or Variant value. If you are converting to a nonstring value, you can include some error handling in case the user enters a string that cannot be converted. "XYZ", for example, cannot be converted to a numeric value.

You can use the Print statement or the MessageBox function or statement to display a message to the user. The Print statement displays the message in the current output window, which varies depending on the Lotus product in which you are working. MessageBox displays a message box. The message box contains an optional title, the message, an optional icon, and one or more command buttons.

If you simply want to display a message, you can use a MessageBox statement and include an OK button (the default). The user reads the message, clicks OK, and the script proceeds to the next statement.

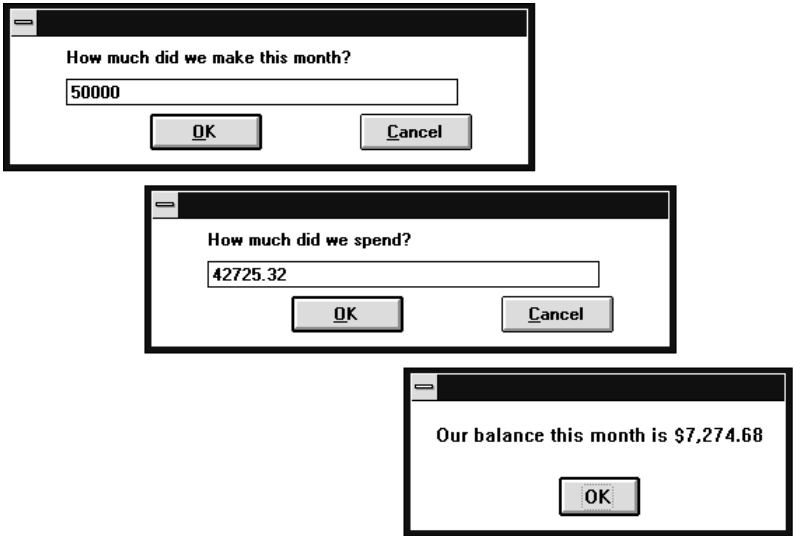
On the other hand, you may want to offer the user two or more options, in which case you use the MessageBox function and include two or more command buttons. For example, you can include OK and Cancel buttons. If the user clicks OK, the MessageBox function returns IDOK (1). If the user clicks Cancel, the function returns IDCANCEL (2). You can use an If statement or Case statement to respond to the user's response accordingly.

The following example uses the InputBox function to get monthly revenue and expenses from the user, converting strings to Currency.

The script computes the balance, then uses a MessageBox statement to display the balance, formatted as Currency.

```
Sub CalcBalance
   Dim revenue As Currency, expenses As Currency, balance As Currency
   revenue@ = CCur(InputBox("How much did we make this month?"))
   expenses@ = CCur(InputBox("How much did we spend?"))
   balance@ = revenue@ - expenses@
   MessageBox "Our balance this month is " _
      & Format(balance@, "Currency")
End Sub
```

Here are the two input boxes with sample entries and the resulting message box:



If the user enters a string that the CCur function cannot convert to Currency, an error condition occurs. You can use an On Error statement to branch to an error-handling routine in such a case.

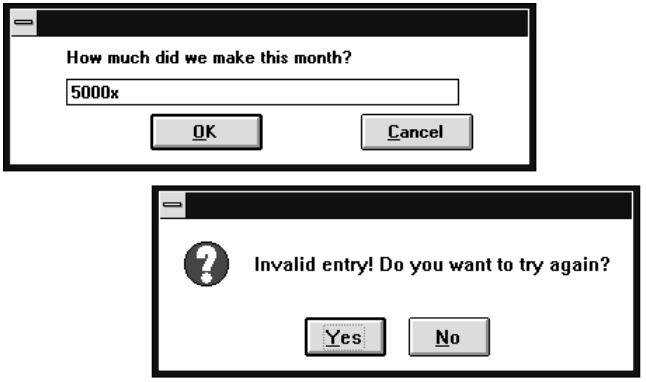
The following expanded version of the example uses the MessageBox function to ask the user whether he or she wants to try again. If the user clicks Yes, the function returns IDYES (6), the script branches back to the EnterValues label, and the user can try again. If the user clicks No, the function returns IDNO (7), and the script exits the CalcBalance sub.

The second message box also contains a question mark icon, specified by MB_ICONQUESTION (32). To use constants rather than the numbers to which they correspond as MessageBox arguments, you must include the file that defines these constants, LSCONST.LSS, in the module declarations.

```
%Include "LSCONST"

Sub CalcBalance
   Dim revenue As Currency, expenses As Currency, balance As Currency
   EnterValues:
   On Error GoTo BadCur:
   revenue@ = CCur(InputBox("How much did we make this month?"))
   expenses@ = CCur(InputBox("How much did we spend?"))
   balance@ = revenue@ - expenses@
   MessageBox "Our balance this month is " _
      & Format(balance@, "Currency")
   Exit Sub
```

```
   BadCur:
   If MessageBox("Invalid entry! Do you want to try again?", _
      MB_YESNO + MB_ICONQUESTION) = IDYES Then GoTo EnterValues
   Exit Sub
End Sub
```

When the user enters an invalid entry, the message box offers the option of making another entry:



For more information about error processing, see Chapter 8, “Error Processing.”

Reading and Writing Files

You can use LotusScript to read and write files. To create a file, you open and write to a file that does not yet exist.

LotusScript provides three modes of file access: sequential (input, output, or append), random, and binary.

Use sequential access to read and write unstructured text files or text files with variable-length records. You can use user-defined data type variables with variable-length string members to read and write variable-length records. Numerical data is stored in the file as text strings.

Use random access for files that contain fixed-length records. You can use the Seek statement and a record number for immediate read or write access to any record in the file. Each record can contain a scalar value or the members of a user-defined data type variable. If the record includes strings, use fixed-length string variables so that each record is the same length.

For a discussion about using user-defined data types to work with files, see “Working with data stored in files” in Chapter 5.

Binary access provides immediate access by number to any byte in the file. In general, you use binary access to read and write bytes of data. You can however, also use binary access to write a stream of characters to an unstructured text file.

Opening files

Use the FreeFile function to get a file number, and then use an Open statement to open a file.

fileNumber% **= FreeFile**
Open *fileName$* [**For** {**Input | Output | Append | Binary | Random** }]
[**Access** {**Read | Read Write | Write**}]
[{**Shared | Lock Read | Lock Read Write | Lock Write** }]]
As *fileNumber%*
[**Len =** *recLen%*]

In the Open statement, you specify access mode and the read and / or write operation you intend to perform. If other processes or users have concurrent access to the file (over a network, for example), you can also specify how the file is to be shared.

For random access, you specify a record length (unless you are using the default of 128 bytes). To determine record length, you can use the Len or LenB function to return the length of the scalar variable or user-defined data type variable you are using to read and / or write records. To enhance performance during sequential access to a file, you can specify a buffer size for the read / write operations.

Reading from files and writing to them

If you open the file for sequential input or append access, you can use the Input function to read a specified number of characters into a String (or Variant) variable. For example, you can use the Input function in conjunction with the LOF function, which returns the length of an open file, to read the entire file (up to 32,000 characters) into a String variable:

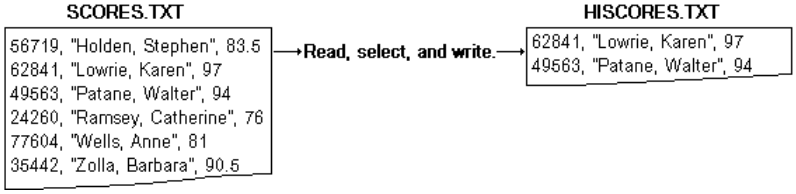
```
fileNumber% = FreeFile
Open "DATA.TXT" For Input As fileNumber%
fileContents$ = Input(LOF(fileNumber%), fileNumber%)
```

To write an extended unstructured string rather than a fixed-length or variable-length record to a text file, you can open the file for binary access and use a Put statement. The following Put statement overwrites the previous contents of a text file starting at the first byte. If the new string is shorter than the previous contents, the Put operation does not overwrite to the end of the file.

```
Open "DATA.TXT" For Binary Access Write As fileNumber%
Put fileNumber%, 1, fileContents$
```

If a file contains variable-length records, use the Input # and Write # statements to read and write records. The Input # statement reads a record into a list of variables, and the Write # statement writes to a file from a list of variables. Write # statements delimit and format entries so that they can be read by Input # statements. In both cases, the list of variables may be the members of a user-defined data type variable.

The following example reads each record from SCORES.TXT into a variable-length user-defined data type variable. If the student's score is at least 92 , the script writes the record to HISCORES.TXT. The process continues until the EOF function returns TRUE (-1), indicating that the script has reached the end of SCORES.TXT.




```
Type Student
   ID As Long
   Name As String ' Variable-length string variable
   Score As Single
End Type
Dim undergrad As Student

Sub WriteGoodStudents
   Dim fileNum1 As Integer, fileNum2 As Integer
   fileNum1% = FreeFile
   Open "SCORES.TXT" For Input As fileNum1%
   fileNum2% = FreeFile
   Open "HISCORES.TXT" For Append As fileNum2%
   While Not EOF(fileNum1%) ' Read until end of file.
     Input #fileNum1%, undergrad.ID, undergrad.Score
     If undergrad.Score > 92 Then
       Write #fileNum2%, undergrad.ID, undergrad.Name, undergrad.Score
     End If
   Wend
   Close fileNum1%
   Close fileNum2%
End Sub
```

You can also use a Print # statement to write to a sequential text file, but Print # does not delimit and format the record to ensure that it can be read with an Input # statement.

When you are using sequential access to write to a file, you can either replace the previous contents of the file (if any) or append to the file. In other words, you can open the file in input mode or append mode. You cannot insert or replace text in the middle of the file.

You can also use the Line Input # statement to read each line into a String variable. Write # and Print # statements put a newline character at the end of each operation, so lines normally correspond to variable-length records (unless you write multiline strings).

When you open a file for random or binary access, the file position is 1 (the first record or byte). Use a Get statement to read data into a variable, and use the Put statement to write data from a variable to the file. The variable may be a user-defined data type variable. Each Get and Put operation advances the file position accordingly. You can use the Seek statement to set the file position to a fixed-length record (random access) or to a byte (binary access). To get the current file position, use the Seek function.

Here is a revision of the preceding example, using fixed-length records and random access. Performance is better, numeric information is stored as such (rather than as strings), but the fixed-length string takes up a little extra space in each record.

```
Type Student
   ID As Long
   Name As String * 20 ' Fixed-length string variable.
   Score As Single
End Type
Dim undergrad As Student

Sub WriteGoodStudents
   Dim fileNum1 As Integer, fileNum2 As Integer
   fileNum1% = FreeFile
   Open "TESTSCORES.TXT" For Random Access Read As fileNum1% _
                         Len = Len(undergrad)
   fileNum2% = FreeFile
   Open "GOODSCORES.TXT" For Random Access Write As fileNum2% _
                          Len = Len(undergrad)
   While Not EOF(fileNum1%) ' Read until end of file.
      Get #fileNum%1,, undergrad
      If undergrad.Score > 92 Then
         Put #fileNum2%,, undergrad
      End If
   Wend
   Close fileNum1%
   Close fileNum2%
End Sub
```

Closing files

As soon as you complete your read / write operations, use the Close statement to close the file. If you modified the file, the Close statement also writes modifications to disk.

You must close the file before you can open it again. If you want to change access mode or operation (from read to write, for example), you must close the file, then open it again.

For more information about working with files, see Chapter 4, "File Handling," in the *LotusScript Language Reference*.

Interacting with Other Programs

LotusScript provides a number of functions and statements that you can use to interact with other programs and the operating system. You can also use Object Linking and Embedding (OLE) and Dynamic Data Exchange (DDE) to incorporate functionality and data from other Windows applications into your LotusScript applications.

Functions and statements for interacting with other programs

LotusScript provides several functions and statements that you can use to interact with other programs and with the operating system.

Function/Statement	*Purpose*
Shell function	Starts another program
ActivateApp function	Activates (gives focus to) the specified window
SendKeys statement	Sends keystrokes to the active window
Environ function	Returns the current value of an environment variable
Yield function/statement	Transfers control during script execution to the operating system

The Windows platform supports all of these functions and statements. On the OS/2®, UNIX®, or Macintosh® platforms, some of these are only partially supported or not supported at all. For more information, see Appendix D, "Platform Differences," in the *LotusScript Language Reference*.

The following example uses all of these functions and statements to interact with a Windows accessory, Notepad:

- The Environ function returns the Windows Temp directory, the directory where Windows creates and maintains temporary files.

  Note On the Windows and OS/2 platforms, the operating system and some programs make use of environment variables that you set. Under MS-DOS®, for example, you use CONFIG.SYS, AUTOEXEC.BAT, and other batch files to set environment variables. You can use the MS-DOS Set command to see a list of environment variables and their current settings. In a script, you can use the Environ function to return the current value of an environment variable.

- The Shell function starts NOTEPAD.EXE.
- The ActivateApp function makes sure that Notepad has the focus so that keystrokes can be sent to it.

- SendKeys statements save a note the user writes in a text file, minimize the Notepad window, and close Notepad.
- The Yield function lets Windows pass control to Notepad so the user can use it to compose a note.

The example contains two module-level variables and four subs. The module-level variables are String variables:

```
Dim startDir As String ' The current directory at startup.
Dim fileName As String ' The note file name.
```

The four subs are Initialize, CreateNote, ReadNote, and Terminate. Initialize automatically executes when the module is loaded. In turn, Initialize calls CreateNote and ReadNote. Terminate executes before the module is unloaded.

The Initialize sub makes the Windows Temp directory the current directory, makes sure that a file named _MYNOTE.EXE exists and is empty, calls the CreateNote sub, then calls the ReadNote sub.

```
Sub Initialize
   Dim tempDir As String, taskID As Integer
   ' Store the name of the current directory, then make the
   ' Windows Temp directory the current directory.
   startDir$ = CurDir$
   tempDir$ = Environ("Temp")
   ChDir tempDir$
   fileName$ = "_MYNOTE.TMP"
   ' Make sure the file exists and is empty before opening Notepad.
   fileNum% = FreeFile
   Open fileName$ For Output As fileNum%
  Write #fileNum% ' The file now contains only an empty line.
  Close fileNum%
  ' Open the file (guaranteed to exist) in Notepad.
  taskID% = Shell("notepad " & fileName$)
   CreateNote ' Create the note. See the CreateNote sub below.
   ReadNote ' Display the note. See the ReadNote sub below.
End Sub
```

The CreateNote sub creates a header for the note, including the current date and time, displays a message, activates (shifts focus to) Notepad, and sends the header to Notepad. Then it yields control to Windows for 10 seconds. Windows, in turn allows the user to type into Notepad. If the 10-second While loop with the Yield were excluded, script execution would continue without any pause, giving the user no time to enter a note.

After the 10 seconds have elapsed, an ActivateApp statement insures that Notepad has the focus (just in case the user has shifted focus to another window), and a SendKeys statement sends keystrokes for the File Save menu command and the Control menu Minimize command.

The keystrokes for File Save are Alt+fs and the keystrokes for Minimize are Alt+spacebar+n. Alt+spacebar+ opens the Control menu in the Notepad title bar. In a SendKeys statement, % represents the Alt key.

```
Sub CreateNote
   Dim header As String, finish As Single
   MessageBox "Write your note."
   header$ = Format(Now, LongDate) &"~~Note: "
   ActivateApp "notepad - " & fileName$
   SendKeys "~" & header$, TRUE ' Send the note header to Notepad.
   finish! = Timer + 10
   While Timer < finish!
      Yield
   Wend
   ActivateApp "notepad - " & fileName$
   SendKeys "%fs% n",TRUE ' Save the file and minimize the window.
End Sub
```

The ReadNote sub displays a message box, opens the file that was just saved, inputs the file contents into a String variable, and displays a message with the contents. The file name appears in the message box title bar.

```
Sub ReadNote
   MessageBox "Read your note."
   fileNum% = FreeFile
   Open fileName$ For Input As #fileNum%
   message$ = Input$(LOF(fileNum%), fileNum%)
   Close fileNum%
   MessageBox message$, , fileName$
End Sub
```

The Terminate sub then executes. Once again, an ActivateApp statement shifts focus to Notepad, just in case the user moved the focus to another window. A SendKeys statement sends Alt+F4 to Notepad, which closes Notepad. Then the sub makes the current directory at startup the current directory again.

```
Sub Terminate
   ActivateApp "notepad - " & fileName$
   SendKeys "%{f4}", TRUE
   ChDir startDir$
End Sub
```

OLE Automation

A Windows application that supports OLE Automation provides a set of product classes, each with a set of properties and methods. You can create and manipulate objects in such an application much as you do in the Lotus product from which you are running LotusScript.

Shapeware® Visio™, for example, is a Windows drawing package that supports OLE automation. The following example builds an array of strings. Each string contains the name and job title of a manager on a Visio organizational chart.

In the module declarations, declare a dynamic one-dimensional array of strings:

```
Dim manager() As String
```

The GetManagers sub uses the CreateObject function to create an instance of the Visio Application class, which runs a new instance of the Visio program (VISIO.EXE). To get an instance that is already running, use the GetObject function.

A Visio Application object contains a collection of documents. Each document contains a collection of pages, and each page contains a collection of shapes. Visio provides a class for each of these: Application, Documents, Document, Pages, Page, Shapes, and Shape.

GetManagers uses the Documents class Open method to open a drawing file, a Document object. The sub then cycles through the pages in the document and the shapes on each page. For each shape with "Manager" in its Name property, the sub places the Text property value in a new element of the array. The Text property for a Manager shape contains a manager's name and job title.

```
Sub GetManagers
   ' Use Variant variables for objects
   Dim appVisioV As Variant, docObjV As Variant
   Dim shapesObjV As Variant, shapeObjV As Variant
   Dim orgchart As String
   Dim iPage As Integer, iShape As Integer, iManager As Integer
   Set appVisioV = CreateObject("visio.application")
   orgchart$ = "c:\visio\drawings\orgchart.vsd"
   Set docObjV = appVisioV.Documents.Open(orgchart$)
   For iPage% = 1 To docObjV.Pages.Count
      Set shapesObjV = docObjV.Pages.Item(iPage%).Shapes
      For iShape% = 1 To shapesObjV.Count
         Set shapeObjV = shapesObjV.Item(iShape%)
         If Instr(shapeObjV.Name, "Manager") > 0 Then
            iManager% = iManager% + 1
            ReDim Preserve manager$(1 To iManager%)
            manager$(iManager%) = shapeObjV.Text
         End If
      Next iShape%
   Next iPage%
   appVisioV.Quit
End Sub
```

To display the contents of the array, use the following:

```
For i% = 1 To Ubound(manager$)
      Print manager$(i%)
Next
```

For information about Visio classes, including their properties and methods, see the Visio documentation.

Dynamic Data Exchange (DDE)

You may be able to use DDE to exchange data between the Lotus product with which you are working and another Windows program. In Lotus Forms, for example, you can create a DDE object in a script and use the object to retrieve data from, poke (send) data to, or send a command to a Windows program that is registered as a DDE server application.

The following Lotus Forms example starts 1-2-3 for Windows and opens a worksheet, then retrieves the value in the range named MiscTotal and places it in field 1 of the current form.

```
Sub BUTTONButton2(B2 As Button)
   Dim taskId As Integer
   ' Start 1-2-3 (the DDE server) and open a worksheet.
   taskId% = Shell(c:\123r5w\programs\123w.exe, _
                  d:\work\expenses.wk4")
   ' Create the DDE object.
   Set DDE123 = New DDE("123worksheet", "d:\work\expenses.wk4")
   ' Retrieve the value in the range named MiscTotal and place it in
   ' Field 1 of the current form.
   Field1.Value = DDE123.Request ("MiscTotal")
   ' Terminate the DDE conversation
   DDE123.Terminate
End Sub
```

For more information about the DDE object, see the Lotus Forms documentation. For more information about using 1-2-3 for Windows as a DDE server, see the 1-2-3 for Windows documentation.

Calling C Functions

In some cases, you may want to use functionality that is provided by a library of C functions. Under Windows, for example, you can use functions in Dynamic Link Libraries. You may obtain such libraries from a variety of sources. All Windows users have access to the libraries in the Windows application programming interface (API).

To work with C functions, you need documentation that explains their input and output parameters, return values, and what operations they actually perform. The Windows Software Developer's Kit, for example, includes Windows API documentation. The Windows API is also documented in a variety of commercially available books.

Suppose, for example, that you want to change the text that appears in the title bar of the window from which you are running LotusScript. You can use the SetWindowText function in the Windows User library to perform this operation.

Declaring C functions

To use C functions, first you must declare them in Declare statements. Declare statements appear at the module level, so enter these statements in the declarations section of the module where you want to call the C functions.

In a Declare statement, you can declare a C function as either a function or a sub. The syntax is:

Declare [**Public** | **Private**] {**Function** | **Sub**}

LSname **Lib** *libName*

[**Alias** *aliasName*]

([*argList*] **)** [**As** *returnType*]

If the C function does not return a value, or you are not interested in the return value, you can declare it as a Sub. In either case, the Declare statement identifies the library containing the function. All the C functions mentioned in this section come from the User library in the Windows 3.1 API.

GetActiveWindow takes no parameters and returns the handle (an integer) of the active window (the window with focus).

```
Declare Function GetActiveWindow Lib "User" () As Integer
```

SetWindowText returns nothing, so you can declare it as a sub. It has two input parameters: the window handle and a string. As long as they are valid LotusScript identifiers, you can use your own parameter names or copy the names used in the API documentation, as in the example below.

```
Declare Sub SetWindowText Lib "User" (ByVal hWnd As Integer, _
                                      ByVal lpString As String)
```

Passing arguments to C functions

By default, LotusScript passes arguments to functions and subs by reference. If the argument is an array, a user-defined data type variable, or an object reference variable, you must pass it by reference. In most other cases, you use the ByVal keyword to pass variables by value.

Passing strings

When you pass strings by value, LotusScript actually creates a NULL-terminated string (which is what the C function expects) and passes a pointer to the string. If you are passing a pointer to something other than a string, then pass the parameter by reference.

Here is a sub that uses the Windows C functions GetActiveWindow and SetWindowText to set the title of the active window (the window with focus):

```
Sub Initialize
   Dim activeWin As Integer, winTitle As String
   activeWin% = GetActiveWindow()
   winTitle$ = "This is my window!"
   SetWindowText activeWin%, winTitle$
End Sub
```

To get a window title at run time, use the GetWindowText function. GetWindowText has one input parameter (the window handle, and integer in Windows 3.1) and two output parameters: a String variable and a buffer size (the maximum length of the string). The return value is the length of the string that the function places in the String variable.

```
Declare Function GetWindowText Lib "User" _
                    (ByVal hWnd As Integer,
                    ByVal lpString As String  _
                    ByVal chMax As Integer) As Integer
```

You must be careful when working with a String variable that is given a value by a C function. If the C function assigns a value that is larger than the length already allocated for the string, it overwrites memory not allocated for the string. The results are unpredictable and may cause a crash.

You can make sure that the String variable has space for the string in one of two ways:

- Assign it a value that is at least as long as the string to be returned before you pass the variable to the C function.
- Declare it as a sufficiently sized fixed-length String variable.

For example, if the maximum length for the string is 255, then you can use the String function to put 255 NULL characters in a variable-length String variable:

```
winTitle$ = String(255, 0)
```

Or you can declare winTitle as a fixed-length String variable:

```
Dim winTitle As String * 255
```

GetWindowText returns the actual length of the string. If you use a variable-length String variable, you can use the return value to get rid of the padding at the end of the string. For example:

```
Dim winTitle As String, winLength As Integer
winTitle = String(255, 0)
activeWin% = GetActiveWindow()
winTitleLength% = GetWindowText(activeWin%, winTitle$, 255)
winTitle$ = Left(winTitle$, winTitleLength%)
```

Note If you use a C function that does not return the length of a string, you can extract the left portion of the string up to the first NULL character as follows:

```
stringFromC$ = Left(stringFromC$, Instr(stringFromC$, Chr$(0)) -1)
```

Using user-defined data type variables

The GetWindowRect C function uses a structured type to retrieve the screen coordinates (in pixels) of the specified window. You must use a Type statement to define the structure. GetWindowRect does not have a return value, so you can declare it as a sub. You pass the window handle by value and the user-defined data type variable by reference. The window handle is an input parameter (it identifies the window), and the Rect user-defined data type variable is an output parameter (GetWindowRect sets its values).

The following set of declarations also includes MoveWindow, which you can use to move and/or rezize the window. This example also uses data type suffix characters to save space in the Declare statements.

```
Type Rect
   left As Integer
   top As Integer
   right As Integer
   bottom As Integer
End Type
Declare Sub GetWindowRect Lib "User" (ByVal hWnd%, lpRect As Rect)

' MoveWindow takes input parameters for the window handle, the
' top left coordinates, the width and height, and a repaint flag.
' The repaint flag (TRUE or FALSE) indicates whether to repaint the
' the window after the move/resize operation.
Declare Sub MoveWindow Lib "User" (ByVal hWnd%, ByVal nLeft%, _
      ByVal nTop%, ByVal nWidth%, ByVal nHeight%, ByVal fRepaint%)
Declare Function GetActiveWindow Lib "User" () As Integer
```

```
Sub Initialize
' Cut the width and height of the active window in half, keeping the
' same coordinates for the top left corner.
   Dim activeWin As Integer, winRect As Rect
   activeWin% = GetActiveWindow()
   GetWindowRect activeWin%, winRect
   MoveWindow activeWin%, winRect.left, winRect.top, _
              winRect.Right/2, winRect.bottom/2, TRUE
End Sub
```

Extended example

The following example uses five Windows 3.1 API functions. The user identifies a window in which to work. The script finds the window, resets the window text, and yields control as long as the user keeps the focus in the window. When the user moves focus out of the window, the script restores the original window text and displays a message. If the user asked for a window that does not exist or is not running, the script also displays an appropriate message.

All declarations are at the module level.

```
' Gets the handle of the active window.
Declare Function GetActiveWindow Lib "User" () As Integer

' Gets the handle of the next window.
Declare Function GetNextWindow Lib "User" _
                   (ByVal hwnd As Integer, _
                   ByVal uFlag As Integer)
                   As Integer
' Windows constant for uFlag parameter: return the handle of the next
' (not the previous) window in the window manager's list.
Const GW_HWNDNEXT =2

' Makes a window (identified by its handle) the active window.
Declare Sub SetActiveWindow Lib "User" (ByVal hwnd As Integer)

' Gets the text in the window title bar.
Declare Function GetWindowText Lib "User" _
                   (ByVal hwnd As Integer,
                   ByVal lpString As String, _
                   ByVal chMax As Integer) As Integer

' Sets the text in the window title bar.
Declare Sub SetWindowText Lib "User" _
                   (ByVal hwnd As Integer, _
                   ByVal lpString$)
```

```
Sub Initialize
   Dim winTitle As String, winTitleLength As Integer
   ' Put 255 NULLs in winTitle$, so enough space is allocated for the
   ' GetWindowText output string.
   winTitle$ = String(255, 0)
   Dim findWinTitle As String, tempWinTitle As String
   Dim curWindow As Integer, found As Integer
   tempWinTitle$ = "I'm working here now!"
   findWinTitle$ = InputBox("What window do you want to use?")
   ' If the input box is empty, exit the sub.
   If Len(findWinTitle$) = 0 Then Exit Sub
   ' Get the handle of the active window (the window from which this
   ' script is running).
   curWin% = GetActiveWindow%()
   ' Search for the window the user indicated. The search continues
   ' until the window is found or all windows have been examined.
   ' GetNextWindow returns 0 after it has cycled through all windows.
   Do While curWin% <> 0
      ' Get the next window.
      curWin% = GetNextWindow(activeWin%, GW_HWNDNEXT)
      ' Get the window title.
      winTitleLength% = GetWindowText(curWin%, winTitle$, 255)
      ' If text the user entered is part of the window title
      ' (do a case-insensitive text comparison), then the search
      ' is done. Otherwise, continue the Do loop.
      If Instr(1, winTitle$, findWinTitle$, 1) > 0 Then
         found% = TRUE
         Exit Do
      End If
```

```
 Loop
   ' If the window was found, then reset the window title, and make it
   ' the active window.
   If found% = TRUE Then
      SetWindowText curWin%, tempWinTitle$
      SetActiveWindow(curWin%)
      ' As long as it remains the active window, yield control,
      ' letting the user continue to work in the window.
      While GetActiveWindow%() = curWin%
         Yield
      Wend
      ' The user moved focus out of the window.
      ' Get rid of the padding at the end of the String variable.
      winTitle$ = Left(winTitle$, winTitleLength%)
      ' Display a message, and set the window title back to its
      ' original.
      MessageBox "Done working with " & winTitle$ & "!"
      SetWindowText curWin%, winTitle$
   ' If the window was not found, display a message.
   Else
      MessageBox "Window not found!"
   End If
End Sub
```

Index

Symbols

A

B

C

F

G

H

I

O

P

Q

R

S

T

U

V

W

X

Y